

Computational Linguistics

ELIZA—A Computer Program For the Study of Natural Language Communication Between Man And Machine

JOSEPH WEIZENBAUM
Massachusetts Institute of Technology,* Cambridge, Mass.

ELIZA is a program operating within the MAC time-sharing system at MIT which makes certain kinds of natural language conversation between man and computer possible. Input sentences are analyzed on the basis of decomposition rules which are triggered by key words appearing in the input text. Responses are generated by reassembly rules associated with selected decomposition rules. The fundamental technical problems with which ELIZA is concerned are: (1) the identification of key words, (2) the discovery of minimal context, (3) the choice of appropriate transformations, (4) generation of responses in the absence of key words, and (5) the provision of an editing capability for ELIZA "scripts". A discussion of some psychological issues relevant to the ELIZA approach as well as of future developments concludes the paper.

Introduction

It is said that to explain is to explain away. This maxim is nowhere so well fulfilled as in the area of computer programming, especially in what is called heuristic programming and artificial intelligence. For in those realms machines are made to behave in wondrous ways, often sufficient to dazzle even the most experienced observer. But once a particular program is unmasked, once its inner workings are explained in language sufficiently plain to induce understanding, its magic crumbles away; it stands revealed as a mere collection of procedures, each quite comprehensible. The observer says to himself "I could have written that". With that thought he moves the program in question from the shelf marked "intelligent", to that reserved for curios, fit to be discussed only with people less enlightened than he.

Work reported herein was supported (in part) by Project MAC, an MIT research program sponsored by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research Contract Number Nonr-4102-011.

* Department of Electrical Engineering.

A. G. OETTINGER, Editor

The object of this paper is to cause just such a re-evaluation of the program about to be "explained". Few programs ever needed it more.

ELIZA Program

ELIZA is a program which makes natural language conversation with a computer possible. Its present implementation is on the MAC time-sharing system at MIT. It is written in MAD-SLIP [4] for the IBM 7094. Its name was chosen to emphasize that it may be incrementally improved by its users, since its language abilities may be continually improved by a "teacher". Like the Eliza of Pygmalion fame, it can be made to appear even more civilized, the relation of appearance to reality, however, remaining in the domain of the playwright.

For the present purpose it is sufficient to characterize the MAC system as one which permits an individual to operate a full scale computer from a remotely located typewriter. The individual operator has the illusion that he is the sole user of the computer complex, while in fact others may be "time-sharing" the system with him. What is important here is that the computer can read messages typed on the typewriter and respond by writing on the same instrument. The time between the computer's receipt of a message and the appearance of its response is a function of the program controlling the dialogue and of such MAC system parameters as the number of users currently corresponding with the system. These latter parameters generally contribute so little to the overall response time that conversational interaction with the computer need never involve truly intolerable delays.

When in conversation with ELIZA, the user types in some statement or set of statements in natural language using normal punctuation and sentence structures. Only the question mark may not be used, because it is interpreted as a line delete character by the MAC system. The user's statement is terminated by a double carriage return which serves to turn control over to ELIZA. ELIZA then analyzes the user's statement and generates some response which it types out. Control is then again with the user.

A typical conversation is the following:

Men are all alike.
IN WHAT WAY
They're always bugging us about something or other.
CAN YOU THINK OF A SPECIFIC EXAMPLE
Well, my boyfriend made me come here.
YOUR BOYFRIEND MADE YOU COME HERE
He says I'm depressed much of the time.
I AM SORRY TO HEAR YOU ARE DEPRESSED

It's true. I am unhappy.
DO YOU THINK COMING HERE WILL HELP YOU NOT TO BE UNHAPPY

I need some help, that much seems certain.
WHY WOULD IT MEAN TO YOU IF YOU GOT SOME HELP
Perhaps you could learn to get along with my mother.
TELL ME MORE ABOUT YOUR FAMILY

My mother takes care of me.
WHO ELSE IN YOUR FAMILY TAKES CARE OF YOU
My father.

YOUR FATHER
You are like my father in some ways.
WHAT RESEMBLANCE DO YOU SEE
You are not very aggressive but I think you don't want me to notice that.

WHY DO YOU THINK I AM NOT VERY AGGRESSIVE
You are afraid of me.

WHY DO YOU THINK I DON'T ARGUE WITH YOU
You are afraid of me.

DOES IT PLEASE YOU TO BELIEVE I AM AFRAID OF YOU
My father is afraid of everybody.

WHAT ELSE COMES TO MIND WHEN YOU THINK OF YOUR FATHER

Bullies.
DOES THAT HAVE ANYTHING TO DO WITH THE FACT THAT YOUR BOYFRIEND MADE YOU COME HERE

The regularized lines are the machine responses.

The gross procedure of the program is quite simple; the text is read and inspected for the presence of a *keyword*. If such a word is found, the sentence is transformed according to a *rule* associated with the keyword, if not a content-free remark or, under certain conditions, an earlier transformation is retrieved. The text so computed or retrieved is then printed out.

In detail, of course, the procedure sketched above is considerably more complex. Keywords, for example, may have a RANK or precedence number. The procedure is sensitive to such numbers in that it will abandon a keyword already found in the left-to-right scan of the text in favor of one having a higher rank. Also, the procedure recognizes a comma or a period as a delimiter. Whenever either one is encountered and a keyword has already been found, all subsequent text is deleted from the input message. If no key had yet been found the phrase or sentence to the left of the delimiter (as well as the delimiter itself) is deleted. As a result, only single phrases or sentences are ever transformed.

Keywords and their associated transformation¹ rules constitute the SCRIPT for a particular class of conversation. An important property of ELIZA is that a script is data; i.e., it is not part of the program itself. Hence, ELIZA is not restricted to a particular set of recognition patterns or responses, indeed not even to any specific language. ELIZA scripts exist (at this writing) in Welsh and German as well as in English.

The fundamental technical problems with which ELIZA must be preoccupied are the following:

(1) The identification of the "most important" keyword

¹The word "transformation" is used in its generic sense rather than that given it by Harris and Chomsky in linguistic contexts.

occurring in the input message.

(2) The identification of some minimal context within which the chosen keyword appears; e.g., if the keyword is "you", is it followed by the word "are" (in which case an assertion is probably being made).

(3) The choice of an appropriate transformation rule and, of course, the making of the transformation itself.

(4) The provision of mechanism that will permit ELIZA to respond "intelligently" when the input text contained no keywords.

(5) The provision of machinery that facilitates editing, particularly extension, of the script on the script writing level.

There are, of course, the usual constraints dictated by the need to be economical in the use of computer time and storage space.

The central issue is clearly one of text manipulation, and at the heart of that issue is the concept of the *transformation rule* which has been said to be associated with certain keywords. The mechanisms subsumed under the slogan "transformation rule" are a number of SLIP functions which serve to (1) decompose a data string according to certain criteria, hence to test the string as to whether it satisfies these criteria or not, and (2) to reassemble a decomposed string according to certain assembly specifications.

While this is not the place to discuss these functions in all their detail (or even to reveal their full power and generality), it is important to the understanding of the operation of ELIZA to describe them in some detail.

Consider the sentence "I am very unhappy these days". Suppose a foreigner with only a limited knowledge of English but with a very good ear heard that sentence spoken but understood only the first two words "I am". Wishing to appear interested, perhaps even sympathetic, he may reply "How long have you been very unhappy these days?" What he must have done is to apply a kind of template to the original sentence, one part of which matched the two words "I am" and the remainder isolated the words "very unhappy these days". He must also have a reassembly kit specifically associated with that template, one that specifies that any sentence of the form "I am BLAH" can be transformed to "How long have you been BLAH", independently of the meaning of BLAH. A somewhat more complicated example is given by the sentence "It seems that you hate me". Here the foreigner understands only the words "you" and "me"; i.e., he applies a template that decomposes the sentence into the four parts:

(1) It seems that (2) you (3) hate (4) me

of which only the second and fourth parts are understood. The reassembly rule might then be "What makes you think I hate you"; i.e., it might throw away the first component, translate the two known words ("you" to "I" and "me" to "you") and tack on a stock phrase (What makes you think) to the front of the reconstruction.

A formal notation in which to represent the decomposition template is:

(0 YOU 0 ME)

and the reassembly rule

(WHAT MAKES YOU THINK I3 YOU.)

The "0" in the decomposition rule stands for "an indefinite number of words" (analogous to the indefinite dollar sign of COMIT) [6] while the "3" in the reassembly rule indicates that the third component of the subject decomposition is to be inserted in its place. The decomposition rule

(0 YOU 1 ME)

would have worked just as well in this specific example. A nonzero integer "n" appearing in a decomposition rule indicates that the component in question should consist of exactly "n" words. However, of the two rules shown, only the first would have *matched* the sentence, "It seems you hate and love me," the second failing because there is more than one word between "you" and "me".



Fig. 1. Keyword and rule list structure

In ELIZA the question of which decomposition rules to apply to an input text is of course a crucial one. The input sentence might have been, for example, "It seems that you hate," in which case the decomposition rule (0 YOU 0 ME) would have failed in that the word "ME" would not have been found at all, let alone in its assigned place. Some other decomposition rule would then have to be tried and, failing that, still another until a match could be made or a total failure reported. ELIZA must therefore have a mechanism to sharply delimit the set of decomposition rules which are potentially applicable to a currently active input sentence. This is the keyword mechanism.

An input sentence is scanned from left to right. Each word is looked up in a dictionary of keywords. If a word is identified as a keyword, then (apart from the issue of precedence of keywords) only decomposition rules containing that keyword need to be tried. The trial sequence can even be partially ordered. For example, the decomposition rule (0 YOU 0) associated with the keyword "YOU" (and decomposing an input sentence into (1) all words in front of "YOU", (2) the word "YOU", and (3) all words following "YOU") should be the last one tried since it is bound to succeed.

Two problems now arise. One stems from the fact that

almost none of the words in any given sentence are represented in the keyword dictionary. The other is that of "associating" both decomposition and reassembly rule with keywords. The first is serious in that the determination that a word is not in a dictionary may well require more computation (i.e., time) than the location of a word which is represented. The attack on both problems begins by placing both a keyword and its associated rules on a list. The basic format of a typical key list is the following

$$\begin{aligned} &(K ((D_1) (R_{1,1}) (R_{1,2}) \dots (R_{1,m_1})) \\ &\quad ((D_2) (R_{2,1}) (R_{2,2}) \dots (R_{2,m_2})) \\ &\quad \vdots \\ &\quad ((D_n) (R_{n,1}) (R_{n,2}) \dots (R_{n,m_n}))) \end{aligned}$$

where K is the keyword, D_i the i th decomposition rule associated with K and $R_{i,j}$ the j th reassembly rule associated with the i th decomposition rule.

A common pictorial representation of such a structure is the tree diagram shown in Figure 1. The top level of this structure contains the keyword followed by the names of lists; each one of which is again a list structure beginning with a decomposition rule and followed by reassembly rules. Since list structures of this type have no predetermined dimensionality limitations, any number of decomposition rules may be associated with a given keyword and any number of reassembly rules with any specific decomposition rule. SLIP is rich in functions that sequence over structures of this type efficiently. Hence programming problems are minimized.

An ELIZA script consists mainly of a set of list structures of the type shown. The actual keyword dictionary is constructed when such a script is first read into the hitherto empty program. The basic structural component of the keyword dictionary is a vector KEY of (currently) 128 contiguous computer words. As a particular key list structure is read the keyword K at its top is randomized (hashed) by a procedure that produces (currently) a bit integer "i". The word "always", for example, yields the integer 14. $KEY(i)$, i.e., the i th word of the vector KEY, is then examined to determine whether it contains a list name. If it does not, then an empty list is created, its name placed in $KEY(i)$, and the key list structure in question placed on *that* list. If $KEY(i)$ already contains a list name, then the name of the key list structure is placed on the bottom of the list named in $KEY(i)$. The large dictionary so far attempted contains about 50 keywords. No list named in any of the words of the KEY vector contains more than two key list structures.

Every word encountered in the scan of an input text, i.e., during the actual operations of ELIZA, is randomized by the same hashing algorithm as was originally applied to the incoming keywords, hence yields an integer which points to the only possible list structure which potentially contain that word as a keyword. Even then only the tops of any key list structures that may be found there need be interrogated to determine whether or not a keyword has been found. By virtue of the various

sequencing operations that SLIP makes available, the actual identification of a keyword leaves as its principal product a pointer to the list of decomposition (and hence reassembly) rules associated with the identified keyword. One result of this strategy is that often less time is required to determine whether a given word is not in the keyword dictionary than to locate it if it is there. However, the location of a keyword yields pointers to all information associated with that word.

Some conversational protocols require that certain transformations be made on certain words of the input text independently of any contextual considerations. The first conversation displayed in this paper, for example, requires that first person pronouns be exchanged for second person pronouns and vice versa throughout the input text. There may be further transformations but these minimal substitutions are unconditional. Simple substitution rules ought not to be elevated to the level of transformations, nor should the words involved be forced to carry with them all the structure required for the fully complex case. Furthermore, unconditional substitutions of single words for single words can be accomplished during the text scan itself, not as a transformation of the entire text subsequent to scanning. To facilitate the realization of these desires, any word in the key dictionary, i.e., at the top of a key list structure, may be followed by an equal sign followed by whatever word is to be its substitute. Transformation rules may, but need not, follow. If none do follow such a substitution rule, then the substitution is made on the fly, i.e., during text scanning, but the word in question is not identified as a keyword for subsequent purposes. Of course, a word may be both substituted for and be a keyword as well. An example of a simple substitution is

(YOURSELF = MYSELF).

Neither "yourself" nor "myself" are keywords in the particular script from which this example was chosen.

The fact that keywords can have ranks or precedences has already been mentioned. The need of a ranking mechanism may be established by an example. Suppose an input sentence is "I know everybody laughed at me." A script may contain the word "I" as well as the word "everybody" as a keyword. Without differential ranking, "I" occurring first would determine the transformation to be applied. A typical response might be "You say you know everybody laughed at you." But the important message in the input sentence, begins with the word "everybody". It is very often true that when a person speaks in terms of universals such as "everybody", "always" and "nobody" he is really referring to some quite specific event or person. By giving "everybody" a higher rank than "I", the response "Who in particular are you thinking of?" may be generated.

The specific mechanism employed in ranking is that the rank of every keyword encountered (absence of rank implies rank equals 0) is compared with the rank of the highest ranked keyword already seen. If the rank of the

new word is higher than that of any previously encountered word, the pointer to the transformation rules associated with the new word is placed on top of a list called the keystack, otherwise it is placed on the bottom of the keystack. When the text scan terminates, the keystack has at its top a pointer associated with the highest ranked keyword encountered in the scan. The remaining pointers in the stack may not be monotonically ordered with respect to the ranks of the words from which they were derived, but they are nearly so—in any event they are in a useful and interesting order. Figure 2 is a simplified

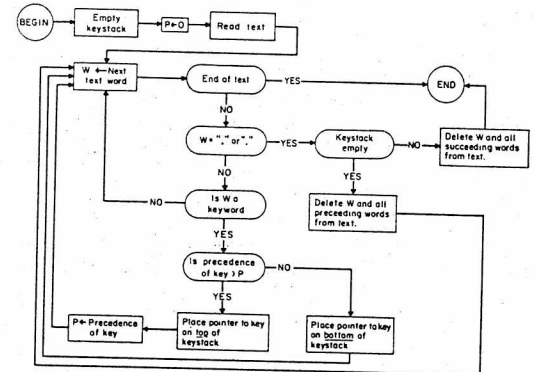


Fig. 2. Basic flow diagram of keyword detection

flow diagram of keyword detection. The rank of a keyword must, of course, also be associated with the keyword. Therefore it must appear on the keyword list structure. It may be found, if at all, just in front of the list of transformation rules associated with the keyword. As an example consider the word "MY" in a particular script. Its keyword list may be as follows:

(MY = YOUR 5 (transformation rules)).

Such a list would mean that whenever the word "MY" is encountered in any text, it would be replaced by the word "YOUR". Its rank would be 5.

Upon completion of a given text scan, the keystack is either empty or contains pointers derived from the keywords found in the text. Each of such pointers is actually a sequence reader—a SLIP mechanism which facilitates scanning of lists—pointing into its particular key list in such a way that one sequencing operation to the right (SEQLR) will sequence it to the first set of transformation rules associated with its keyword, i.e., to the list

((D₁) (R_{1,1}) (R_{1,2}) ... (R₁, R_{m₁})).

The top of *that* list, of course, is a list which serves as a decomposition rule for the subject text. The top of the keystack contains the first pointer to be activated.

The decomposition rule D_1 associated with the keyword K , i.e., $\{(D_1), K\}$, is now tried. It may fail however. For example, suppose the input text was:

You are very helpful.

The keyword, say, is "you", and $\{(D_1), \text{you}\}$ is

$\{0 \text{ I remind you of } 0\}$.

(Recall that the "you" in the original sentence has already been replaced by "I" in the text now analyzed.) This decomposition rule obviously fails to match the input sentence. Should $\{(D_1), K\}$ fail to find a match, then $\{(D_2), K\}$ is tried. Should that too fail, $\{(D_3), K\}$ is attempted, and so on. Of course, the set of transformation rules can be guaranteed to terminate with a decomposition rule which must match. The decomposition rule

$\{0 K 0\}$

will match any text in which the word K appears while

0

will match any text whatever. However, there are other ways to leave a particular set of transformation rules, as will be shown below. For the present, suppose that some particular decomposition rule $\{(D_1), K\}$ has matched the input text. $\{(D_1), K\}$, of course, was found on a list of the form

$\{(D_1)(R_{1,1})(R_{1,2}) \cdots (R_{1,m_1})\}$.

Sequencing the reader which is presently pointing at $\{(D_1), K\}$ will retrieve the reassembly rule $\{(R_{1,i})\}$ which may then be applied to the decomposed input text to yield the output message.

Consider again the input text

You are very helpful

in which "you" is the only key word. The sentence is transformed during scanning to

I are very helpful

$\{(D_1), \text{you}\}$ is " $\{0 \text{ I remind you of } 0\}$ " and fails to match as already discussed. However, $\{(D_2), \text{you}\}$ is " $\{0 \text{ I are } 0\}$ " and obviously matches the text, decomposing it into the constituents

(1) empty (2) I (3) are (4) very helpful.

$\{(R_{2,1}), \text{you}\}$ is

What makes you think I am I

Hence it produces the output text

What makes you think I am very helpful.

Having produced it, the integer 1 is put in front of $\{(R_{2,1})\}$ so that the transformation rule list in question now appears as

$\{(D_2)1(R_{2,1})(R_{2,2}) \cdots (R_{2,m_2})\}$.

Next time $\{(D_2), K\}$ matches an input text, the reassembly rule $\{(R_{2,2})\}$ will be applied and the integer 2 will replace the 1. After $\{(R_{2,m_2})\}$ has been exercised, $\{(R_{2,1})\}$ will again be invoked. Thus, after the system has been in use for a time, every decomposition rule which has matched some input text has associated with it an integer which corresponds to the last reassembly rule used in connection with

that decomposition rule. This mechanism insures that the complete set of reassembly rules associated with a given decomposition rule is cycled through before any repetition occurs.

The system described so far is essentially one which selects a decomposition rule for the highest ranking keyword found in an input text, attempts to match that text according to that decomposition rule and, failing to make a match, selects the next reassembly rule associated with the matching decomposition rule and applies it to generate an output text. It is, in other words, a system which, for the highest ranking keyword of a text, selects a specific decomposition and reassembly rule to be used in forming the output message.

Were the system to remain that simple, then keywords that required identical sets of transformation rules would each require that a copy of these transformation rules be associated with them. This would be logically sound but would complicate the task of script writing and would also make unnecessary storage demands. There are therefore special types of decomposition and assembly rules characterized by the appearance of "=" at the top of the rule list. The word following the equal sign indicates which new set of transformation rules is to be applied. For example, the keyword "what" may have associated with it a transformation rule set of the form

$\{(0) (\text{Why do you ask}) (\text{Is that an important question}) \dots\}$

which would apply equally well to the keywords "how" and "when". The entire keyword list for "how" may therefore be

$\{(\text{How} (= \text{What}))\}$

The keywords "how", "what" and "when" may thus be made to form an equivalence class with respect to the transformation rules which are to apply to them.

In the above example the rule " $(= \text{what})$ " is in the place of a decomposition rule, although it causes no decomposition of the relevant text. It may also appear, however, in the place of a reassembly rule. For example, the keyword "am" may have among others the following transformation rule set associated with it:

$\{(0 \text{ are you } 0) (\text{Do you believe you are } 4) \dots (= \text{what})\}$

(It is here assumed that "are" has been substituted for "am" and "you" for "I" in the initial text scan.) Then the input text

Am I sick

would elicit either

Do you believe you are sick

or

Why do you ask

depending on how many times the general form has already occurred.

Under still other conditions it may be desirable

perform a preliminary transformation on the input text before subjecting it to the decompositions and reassemblies which finally yield the output text. For example, the keyword "you're" should lead to the transformation rules associated with "you" but should first be replaced by a word in the dictionary. The dictionary entry for "you're" is therefore:

$\text{you're} = \text{I'm } \{(0 \text{ I'm } 0) (\text{PRE } (\text{I AM } 3) (= \text{YOU}))\}$

which has the following effect:

(1) Wherever "you're" is found in the input text, it is replaced by "I'm".

(2) If "you're" is actually selected as the regnant keyword, then the input text is decomposed into three constituent parts, namely, all text in front of the first occurrence of "I'm", the word "I'm" itself, and all text following the first occurrence of "I'm".

The first constituent is then reassembled beginning with the code "PRE" is encountered and the decomposed text reassembled such that the words "I AM" appear in front of the third constituent determined by the earlier decomposition.

(4) Control is transferred, so to speak, to the transformation rules associated with the keyword "you", where further decompositions etc. are attempted.

It is to be noted that the set

$\{(0 \text{ I AM } 3) (= \text{YOU})\}$

is logically in the place of a reassembly rule and may therefore be one of many reassembly rules associated with the given decomposition.

Another form of reassembly rule is

$\{(\text{NEWKEY})\}$

which serves the case in which attempts to match on the currently regnant keyword are to be given up and the entire decomposition and reassembly process is to start again on the basis of the keyword to be found in the keystack. Whenever this rule is invoked, the top of the keystack is "popped up" once, i.e., the new regnant keyword recovered and removed from the keystack, and the entire process reinitiated as if the initial text scan had just terminated. This mechanism makes it possible to, in effect, test on key phrases as opposed to single key words.

A serious problem which remains to be discussed is the reaction of the system in case no keywords remain to serve as transformation triggers. This can arise either in case the keystack is empty when NEWKEY is invoked or when the input text contained no keywords initially.

The simplest mechanism supplied is in the form of the special reserved keyword "NONE" which must be part of any script. The script writer must associate the universally matching decomposition rule (0) with it and follow this by as many content-free remarks in the form of transformation rules as he pleases. (Examples are: "Please go on", "That's very interesting" and "I see".)

There is, however, another mechanism which causes the system to respond more spectacularly in the absence of a key. The word "MEMORY" is another reserved pseudo-keyword. The key list structure associated with it differs

from the ordinary one in some respects. An example illuminates this point.

Consider the following structure:

$\{(\text{MEMORY MY})\}$

$\{0 \text{ YOUR } 0 = \text{LETS DISCUSS FURTHER WHY YOUR } 3\}$

$\{0 \text{ YOUR } 0 = \text{EARLIER YOU SAID YOUR } 3\}$

$\vdots$

The word "MY" (which must be an ordinary keyword as well) has been selected to serve a special function. Whenever it is the highest ranking keyword of a text one of the transformations on the MEMORY list is randomly selected, and a copy of the text is transformed accordingly. This transformation is stored on a first-in-first-out stack for later use. The ordinary processes already described are then carried out. When a text without keywords is encountered later and a certain counting mechanism is in a particular state and the stack in question is not empty, then the transformed text is printed out as the reply. It is, of course, also deleted from the stack of such transformations.

The current version of ELIZA requires that one keyword be associated with MEMORY and that exactly four transformations accompany that word in that context. (An application of a transformation rule of the form

$\{(\text{LEFT HAND SIDE} = \text{RIGHT HAND SIDE})\}$

is equivalent to the successive application of the two forms

$\{(\text{LEFT HAND SIDE}), (\text{RIGHT HAND SIDE})\}$.)

Three more details will complete the formal description of the ELIZA program.

The transformation rule mechanism of SLIP is such that it permits tagging of words in a text and their subsequent recovery on the basis of one of their tags. The keyword "MOTHER" in ELIZA, for example, may be identified as a noun and as a member of the class "family" as follows:

$\{(\text{MOTHER DLIST } (/ \text{NOUN FAMILY}))\}$.

Such tagging in no way interferes with other information (e.g., rank or transformation rules) which may be associated with the given tag word. A decomposition rule may contain a matching constituent of the form $\{(/ \text{TAG1 TAG2 } \dots)\}$ which will match and isolate a word in the subject text having any one of the mentioned tags. If, for example, "MOTHER" is tagged as indicated and the input text

"CONSIDER MY AGED MOTHER AS WELL AS ME"

subjected to the decomposition rule

$\{0 \text{ YOUR } 0 (/ \text{FAMILY } 0)\}$

(remembering that "MY" has been replaced by "YOUR"), then the decomposition would be

(1) CONSIDER (2) YOUR (3) AGED (4) MOTHER
(5) AS WELL AS ME.

Another flexibility inherent in the SLIP text manipulation mechanism underlying ELIZA is the ordering of matching criteria is permitted in decomposition rules. The above input text would have been decomposed

precisely as stated above by the decomposition rule:

(0 YOUR 0 (FATHER MOTHER) 0)

which, by virtue of the presence of "*" in the sublist structure seen above, would have isolated either the word "FATHER" or "MOTHER" (in that order) in the input text, whichever occurred first after the first appearance of the word "YOUR".

Finally, the script writer must begin his script with a list, i.e., a message enclosed in parentheses, which contains the statement he wishes ELIZA to type when the system is first loaded. This list may be empty.

Editing of an ELIZA script is achieved via appeal to a contextual editing program (ED) which is part of the MAC library. This program is called whenever the input text to ELIZA consists of the single word "EDIT". ELIZA then puts itself in a so-called dormant state and presents the then stored script for editing. Detailed description of ED is out of place here. Suffice it to say that changes, additions and deletions of the script may be made with considerable efficiency and on the basis of entirely contextual cues, i.e., without resort to line numbers or any other artificial devices. When editing is completed, ED is given the command to FILE the revised script. The new script is then stored on the disk and read into ELIZA. ELIZA then types the word "START" to signal that the conversation may resume under control of the new script.

An important consequence of the editing facility built into ELIZA is that a given ELIZA script need not start out to be a large, full-blown scenario. On the contrary, it should begin as a quite modest set of keywords and transformation rules and permitted to be grown and molded as experience with it builds up. This appears to be the best way to use a truly interactive man-machine facility—i.e., not as a device for rapidly debugging a code representing a fully thought out solution to a problem, but rather as an aid for the exploration of problem solving strategies.

Discussion

At this writing, the only serious ELIZA scripts which exist are some which cause ELIZA to respond roughly as would certain psychotherapists (Rogerians). ELIZA performs best when its human correspondent is initially instructed to "talk" to it, via the typewriter of course, just as one would to a psychiatrist. This mode of conversation was chosen because the psychiatric interview is one of the few examples of categorized dyadic natural language communication in which one of the participating pair is free to assume the pose of knowing almost nothing of the real world. If, for example, one were to tell a psychiatrist "I went for a long boat ride" and he responded "Tell me about boats", one would not assume that he knew nothing about boats, but that he had some purpose in so directing the subsequent conversation. It is important to note that this assumption is one made by the speaker. Whether it is realistic or not is an altogether separate question. In any case, it has a crucial psychological utility

in that it serves the speaker to maintain his sense of being heard and understood. The speaker further defends his impression (which even in real life may be illusory) by attributing to his conversational partner all sorts of background knowledge, insights and reasoning ability. But again, these are the *speaker's* contribution to the conversation. They manifest themselves inferentially in the *interpretation* he makes of the offered responses. From the purely technical programming point of view then, the psychiatric interview form of an ELIZA script has the advantage that it eliminates the need of storing *explicit* information about the real world.

The human speaker will, as has been said, contribute much to clothe ELIZA'S responses in vestments of plausibility. But he will not defend his illusion (that he is being understood) against all odds. In human conversation a speaker will make certain (perhaps generous) assumptions about his conversational partner. As long as it remains possible to interpret the latter's responses consistently with those assumptions, the speaker's image of his partner remains unchanged, in particular, undamaged. Responses which are difficult to so interpret may well result in an enhancement of the image of the partner, in additional rationalizations which then make more complicated interpretations of his responses reasonable. When, however, such rationalizations become too massive and even self-contradictory, the entire image may crumble and be replaced by another ("He is not, after all, as smart as I thought he was"). When the conversational partner is a machine (the distinction between machine and program is here not useful) then the idea of *credibility* may well be substituted for that of *plausibility* in the above.

With ELIZA as the basic vehicle, experiments may be set up in which the subjects find it credible to believe that the responses which appear on his typewriter are generated by a human sitting at a similar instrument in another room. How must the script be written in order to maintain the credibility of this idea over a long period of time? How can the performance of ELIZA be systematically degraded in order to achieve controlled and predictable thresholds of credibility in the subject? What, in all this, is the role of the initial instruction to the subject? On the other hand, suppose the subject is told he is communicating with a machine. What is he led to believe about the machine as a result of his conversational experience with it? Some subjects have been very hard to convince that ELIZA (with its present script) is *not* human. This is a striking form of Turing's test. What experimental design would make it more nearly rigorous and airtight?

The whole issue of the credibility (to humans) of machine output demands investigation. Important decisions increasingly tend to be made in response to computer output. The ultimately responsible human interpreter of "What the machine says" is, not unlike the correspondent with ELIZA, constantly faced with the need to make credibility judgments. ELIZA shows, nothing else, how easy it is to create and maintain an illusion of understanding, hence perhaps of judgment

deserving of credibility. A certain danger lurks there.

The idea that the present ELIZA script contains *no* information about the real world is not entirely true. For example, the transformation rules which cause the input

Everybody hates me

to be transformed to

... you think of anyone in particular

and other such are based on quite specific hypotheses about the world. The whole script constitutes, in a loose way, a model of certain aspects of the world. The act of writing a script is a kind of programming act and has all the advantages of programming, most particularly that it clearly shows where the programmer's understanding and command of his subject leaves off.

Part of whatever elegance may be credited to ELIZA lies in the fact that ELIZA maintains the illusion of understanding with so little machinery. But there are bounds on the extendability of ELIZA's "understanding" power, which are a function of the ELIZA program itself and not a function of any script it may be given. The crucial test of understanding, as every teacher should know, is not the subject's ability to continue a conversation, but to draw valid conclusions from what he is being told. In order for a computer program to be able to do that, it must at least have the capacity to store *selected* parts of its inputs. ELIZA throws away each of its inputs, except for those few transformed by means of the MEMORY machinery. Of course, the problem is more than one of storage. A great part of it is, in fact, subsumed under the word "selected" used just above. ELIZA in its use so far has had as one of its principal objectives the *concealment* of its lack of understanding. But to encourage its conversational partner to offer inputs from which it can glean medial information, it must *reveal* its misunderstanding. A switch of objectives from the concealment to the revelation of misunderstanding is seen as a precondition to making an ELIZA-like program the basis for an effective natural language man-machine communication system.

One goal for an augmented ELIZA program is thus a system which already has access to a store of information about some aspects of the real world and which, by means of conversational interaction with people, can reveal both what it knows, i.e., behave as an information retrieval system, and where its knowledge ends and needs to be augmented. Hopefully the augmentation of its knowledge will also be a direct consequence of its conversational experience. It is precisely the prospect that such a program will converse with many people and learn something from each of them, which leads to the hope that it will prove an interesting and even useful conversational partner.

One way to state a slightly different intermediate goal is to say that ELIZA should be given the power to slowly build a model of the subject conversing with it. If the subject mentions that he is not married, for example, and later speaks of his wife, then ELIZA should be able to

make the tentative inference that he is either a widower or divorced. Of course, he could simply be confused. In the long run, ELIZA should be able to build up a *belief structure* (to use Abelson's phrase) of the subject and on that basis detect the subject's rationalizations, contradictions, etc. Conversations with such an ELIZA would often turn into arguments. Important steps in the realization of these goals have already been taken. Most notable among these is Abelson's and Carroll's work on simulation of belief structures [1].

The script that has formed the basis for most of this discussion happens to be one with an overwhelmingly psychological orientation. The reason for this has already been discussed. There is a danger, however, that the example will run away with what it is supposed to illustrate. It is useful to remember that the ELIZA program itself is merely a translating processor in the technical programming sense. Gorn [2] in a paper on language systems says:

Given a language which already possesses semantic content, then a translating processor, even if it operates only syntactically, generates corresponding expressions of another language to which we can attribute as "meanings" (possibly multiple—the translator may not be one to one) the "semantic intents" of the generating source expressions; whether we find the result consistent or useful or both is, of course, another problem. It is quite possible that by this method the same syntactic object language can be usefully assigned multiple meanings for each expression...

It is striking to note how well his words fit ELIZA. The "given language" is English as is the "other language", expressions of which are generated. In principle, the given language could as well be the kind of English in which "word problems" in algebra are given to high school students and the other language, a machine code allowing a particular computer to "solve" the stated problems. (See Bobrow's program STUDENT [3].)

The intent of the above remarks is to further rob ELIZA of the aura of magic to which its application to psychological subject matter has to some extent contributed. Seen in the coldest possible light, ELIZA is a translating processor in Gorn's sense; however, it is one which has been especially constructed to work well with natural language text.

REFERENCES

1. ABELSON, R. P., AND CARROLL, J. D. Computer simulation of individual belief systems. *Amer. Behav. Sci.* 9 (May 1965), 24-30.
2. GORN, S. Semiotic relationships in ambiguously stratified language systems. Paper presented at Int. Colloq. Algebraic Linguistics and Automatic Theory, Hebrew U. of Jerusalem, Aug. 1964.
3. BOBROW, D. G. Natural language input for a computer problem solving system. Doctoral thesis, Math. Dept., MIT, Cambridge, Mass., 1964.
4. WEIZENBAUM, J. Symmetric list processor. *Comm. ACM* 6, (Sept. 1963), 524-544.
5. ROGERS, C. *Client Centered Therapy: Current Practice, Implications and Theory*. Houghton Mifflin, Boston, 1951.
6. YNGVE, J. *COMIT Programming Manual*. MIT Press, Cambridge, Mass., 1961.

HOW DO YOU DO. PLEASE TELL ME YOUR PROBLEM)
START
(SORY ((0) (PLEASE DON'T APOLOGIZE)
(APOLOGIES ARE NOT NECESSARY) (WHAT FEELINGS
'O YOU HAVE WHEN YOU APOLOGIZE) (I'VE TOLD YOU
THAT APOLOGIES ARE NOT REQUIRED)))
(DON'T = DON'T)
(CAN'T = CAN'T)
(WON'T = WON'T)
(REMEMBER 5
(DO YOU REMEMBER 0) (DO YOU OFTEN THINK OF 4)
(DOES THINKING OF 4 BRING ANYTHING ELSE TO MIND)
(WHAT ELSE DO YOU REMEMBER)
(WHY DO YOU REMEMBER 4 JUST NOW)
(WHAT IN THE PRESENT SITUATION REMINDS YOU OF 4)
(WHAT IS THE CONNECTION BETWEEN ME AND 4))
(DO TO I REMEMBER 3) (DID YOU THINK I WOULD FORGET 3)
(WHY DO YOU THINK I SHOULD RECALL 5 NOW)
(WHAT ABOUT 3) (-WHAT) (YOU MENTIONED 3))
(0) (NEWKEY)))
(IF 3 ((0 IF 0) (DO YOU THINK ITS LIKELY THAT 3) (DO YOU WISH THAT 3)
(WHAT DO YOU THINK ABOUT 3) (REALLY, 2 3)))
(WHAT DO YOU THINK ABOUT 3) (REALLY, 2 3)))
(DREAMT 4 ((0 YOU DREAMT 3)
(REALLY, 4) (HAVE YOU EVER FANTASIED 4 WHILE YOU WERE AWAKE)
(HAVE YOU DREAMT 4 BEFORE) (-DREAM) (NEWKEY)))
(DREAMED = DREAMT 4 (-DREAM))
(DREAM 3 ((0) (WHAT DOES THAT DREAM SUGGEST TO YOU)
(DO YOU DREAM OFTEN) (WHAT PERSONS APPEAR IN YOUR DREAMS)
(DON'T YOU BELIEVE THAT DREAM HAS SOMETHING TO DO WITH
YOUR PROBLEM) (NEWKEY)))
(DREAMS = DREAM 3 (-DREAM))
(HOW (-WHAT))
(WHEN (-WHAT))
(ALIKE 10 (-DIT))
(SAME 10 (-DIT))
(CERTAINLY (=YES))
(FEEL DLIST/(BELIEF))
(THINK DLIST/(BELIEF))
(BELIEVE DLIST/(BELIEF))
(WICH DLIST/(BELIEF))
(MEMORY MY (0 YOUR 0 = LETS DISCUSS FURTHER WHY YOUR 3)
(0 YOUR 0 = EARLIER YOU SAID YOUR 3)
(0 YOUR 0 = BUT YOUR 3)
(0 YOUR 0 = DOES THAT HAVE ANYTHING TO DO WITH THE FACT THAT YOUR 3))
(NONE ((0) (I AM NOT SURE I UNDERSTAND YOU FULLY)
(PLEASE GO ON)
(WHAT DOES THAT SUGGEST TO YOU)
(DO YOU FEEL STRONGLY ABOUT DISCUSSING SUCH THINGS)))
(PERHAPS ((0) (YOU DON'T SEEM QUITE CERTAIN)
(WHY THE UNCERTAIN TONE)
(CAN'T YOU BE MORE POSITIVE)
(YOU AREN'T SURE) (DON'T YOU KNOW)))
(MAYBE (=PERHAPS))
(NAME 15 ((0) (I AM NOT INTERESTED IN NAMES)
(I'VE TOLD YOU BEFORE, I DON'T CARE ABOUT NAMES -
PLEASE CONTINUE)))
(PLEASE CONTINUE)))
(DEUTSCH (=XFREDD))
(FRANCAIS (=XFREMD))
(ITALIANO (=XFREMD))
(ESPANOL (=XFREMD))
(XFREDD ((0) (I AM SORRY, I SPEAK ONLY ENGLISH)))
(HELLO ((0) (HOW DO YOU DO. PLEASE STATE YOUR PROBLEM)))
(COMPUTER 50 ((0) (DO COMPUTERS WORRY YOU)
(WHY DO YOU MENTION COMPUTERS) (WHAT DO YOU THINK MACHINES
HAVE TO DO WITH YOUR PROBLEM) (DON'T YOU THINK COMPUTERS CAN
HELP PEOPLE) (WHAT ABOUT MACHINES WORRIES YOU) (WHAT
DO YOU THINK ABOUT MACHINES)))
(MACHINE 50 (=COMPUTER))
(MACHINES 50 (=COMPUTER))
(COMPUTERS 50 (=COMPUTER))
(AM = ARE ((0 ARE YOU 0) (DO YOU BELIEVE YOU ARE 4)

(HOW YOU WANT TO BE 4) (YOU WISH I WOULD TELL YOU YOU ARE 4)
(WHAT WOULD IT MEAN IF YOU WERE 4) (=WHAT))
(0) (WHY DO YOU SAY 'AM') (I DON'T UNDERSTAND THAT)))
(ARE ((0 ARE I 0)
(WHY ARE YOU INTERESTED IN WHETHER I AM 4 OR NOT)
(HOW YOU PREFER IF I WEREN'T 4) (PERHAPS I AM 4 IN YOUR
FANTASIES) (DO YOU SOMETIMES THINK I AM 4) (=WHAT))
(0 ARE 0) (DID YOU THINK THEY MIGHT NOT BF 3)
(WOULD YOU LIKE IT IF THEY WERE NOT 3) (WHAT IF THEY WERE NOT 3)
(POSSIBLY THEY ARE 3)))
(YOUR = MY ((0 MY 0) (WHY ARE YOU CONCERNED OVER MY 3)
(WHAT ABOUT YOUR OWN 3) (ARE YOU WORRIED ABOUT SOMEONE ELSE'S 3)
(REALLY, MY 3)))
(WAS 2 ((0 WAS YOU 0)
(WHAT IF YOU WERE 4) (DO YOU THINK YOU WERE 4)
(WERE YOU 4) (WHAT WOULD IT MEAN IF YOU WERE 4)
(WHAT DOES '4' SUGGEST TO YOU) (=WHAT))
(0 YOU WAS 0)
(WERE YOU REALLY) (WHY DO YOU TELL ME YOU WERE 4 NOW)
(WERE YOU REALLY) (WHY DO YOU TELL ME YOU WERE 4 NOW)
(PERHAPS I ALREADY KNEW YOU WERE 4))
(0 WAS I 0) (WOULD YOU LIKE TO RELIEVE I WAS 4)
(WHAT SUGGESTS THAT I WAS 4)
(WHAT DO YOU THINK) (PERHAPS I WAS 4)
(WHAT IF I HAD BEEN 4))
(0) (NEWKEY)))
(WERE = WAS (=WAS))
(ME = YOU)
(YOU'RE = I'M ((0 I'M 0) (PRE (I ARE 3) (=YOU)))
(I'M = YOU'RE ((0 YOU'RE 0) (PRE (YOU ARE 3) (=I)))
(MYSELF = YOURSELF)
(YOURSELF = MYSELF)
(MOTHER DLIST(/NOUN FAMILY))
(MOM = MOTHER DLIST(/ FAMILY))
(DAD = FATHER DLIST(/ FAMILY))
(FATHER DLIST(/NOUN FAMILY))
(SISTER DLIST(/FAMILY))
(BROTHER DLIST(/FAMILY))
(WIFE DLIST(/FAMILY))
(CHILDREN DLIST(/FAMILY))
(I = YOU
(0 YOU (= WANT NEED) 0) (WHAT WOULD IT MEAN TO YOU IF YOU GOT
(WHY DO YOU WANT 4) (SUPPOSE YOU GOT 4 SOON) (WHAT
IF YOU NEVER GOT 4) (WHAT WOULD GETTING 4 MEAN TO
YOU) (WHAT DOES WANTING 4 HAVE TO DO WITH THIS DISCUSSION))
(0 YOU ARE 0 (=SAD UNHAPPY DEPRESSED SICK) 0)
(I AM SORRY TO HEAR YOU ARE 5) (DO YOU THINK COMING HERE
WILL HELP YOU NOT TO BE 5) (I'M SURE ITS NOT PLEASANT TO
BE 5) (CAN YOU EXPLAIN WHAT MADE YOU 5))
(0 YOU ARE 0 (=HAPPY ELATED GLAD BETTER) 0)
(HOW HAVE I HELPED YOU TO BE 5)
(HAS YOUR TREATMENT MADE YOU 5) (WHAT MAKES YOU 5 JUST
NOW) (CAN YOU EXPLAIN WHY YOU ARE SUDDENLY 5))
(0 YOU WAS 0) (=WAS))
(0 YOU WAS 0) (=WAS))
(0 YOU (/BELIEF) YOU 0) (DO YOU REALLY THINK SO) (BUT YOU ARE
NOT SURE YOU 5) (DO YOU REALLY DOUBT YOU 5))
(0 YOU 0 (/BELIEF) 0 I 0) (=YOU))
(0 YOU ARE 0)
(IS IT BECAUSE YOU ARE 4 THAT YOU CAME TO ME)
(HOW LONG HAVE YOU BEEN 4)
(DO YOU BELIEVE IT NORMAL TO BE 4)
(DO YOU ENJOY BEING 4))
(0 YOU (= CAN'T CANNOT) 0) (HOW DO YOU KNOW YOU CAN'T 4)
(HAVE YOU TRIED)
(PERHAPS YOU COULD 4 HOW)
(DO YOU REALLY WANT TO BE ABLE TO 4))
(0 YOU DON'T 0) (DON'T YOU REALLY 4) (WHY DON'T YOU 4)
(DO YOU WISH TO BE ABLE TO 4) (DOES THAT TROUBLE YOU)
(0 YOU FEEL 0) (TELL ME MORE ABOUT SUCH FEELINGS)
(DO YOU OFTEN FEEL 4)
(DO YOU ENJOY FEELING 4)
(OF WHAT DOES FEELING 4 REMIND YOU))
(0 YOU 0 I 0) (PERHAPS IN YOUR FANTASY WE 3 EACH OTHER)
(0 YOU WISH TO 3 ME)
(YOU SEEM TO NEED TO 3 ME)
(DO YOU 3 ANYONE ELSE))

(0) (YOU SAY 1)
(CAN YOU ELABORATE ON THAT) (DO YOU SAY 1 FOR SOME SPECIAL REASON)
(THAT'S QUITE INTERESTING))
(YOU = 1 ((0 I REMIND YOU OF 0) (=DIT)))
((0 I ARE 0) (WHAT MAKES YOU THINK I AM 4)
(DOES IT PLEASE YOU TO BELIEVE I AM 4)
(DO YOU SOMETIMES WISH YOU WERE 4)
(PERHAPS YOU WOULD LIKE TO BE 4))
(WHY DO YOU THINK I 3 YOU)
(WHO DO YOU THINK I 3 YOU - DON'T YOU)
(WHAT MAKES YOU THINK I 3 YOU)
(REALLY, I 3 YOU) (DO YOU WISH TO BELIEVE I 3 YOU)
(REALLY, I 3 YOU) (DO YOU WISH TO BELIEVE I 3 YOU)
(SUPPOSE I DID 3 YOU - WHAT WOULD THAT MEAN)
(DOES SOMEONE ELSE BELIEVE I 3 YOU))
((0 I 0) (WE WERE DISCUSSING YOU - NOT ME)
(OH, I 3) (YOU'RE NOT REALLY TALKING ABOUT ME - ARE YOU)
(WHAT ARE YOUR FEELINGS NOW)))
(YES ((0) (YOU SEEM QUITE POSITIVE) (YOU ARE SURE).
(I SEE) (I UNDERSTAND)))
(DO YOU WANT TO SAY 'NO' JUST TO BE NEGATIVE)
(MAYBE A BIT NEGATIVE) (WHY NOT) (WHY 'NO'))
((0 YOUR 0 (/FAMILY) 2)
(TELL ME MORE ABOUT YOUR FAMILY) (WHO ELSE IN YOUR FAMILY
IS (YOUR 4) (WHAT ELSE COMES TO MIND WHEN YOU THINK OF
YOUR 4))
((0 YOUR 0) (YOUR 3) (WHY DO YOU SAY YOUR 3) (DOES THAT
SUGGEST ANYTHING ELSE WHICH BELONGS TO YOU) (IS IT
IMPORTANT TO YOU THAT 2 3)))
(CAN ((0 CAN I 0) (YOU BELIEVE I CAN 4 AND DON'T YOU) (=WHAT)
(YOU WANT ME TO BE ABLE TO 4) (PERHAPS YOU WOULD LIKE TO BE
ABLE TO 4 YOURSELF)) ((0 CAN YOU 0) (WHETHER OR NOT YOU CAN 4
DEPENDS ON YOU MORE THAN ON ME) (DO YOU WANT TO BE ABLE TO 4)
(PERHAPS YOU DON'T WANT TO 4) (=WHAT)))
(WHY DO YOU ASK) (DOES THAT QUESTION INTEREST YOU)
(WHAT IS IT YOU REALLY WANT TO KNOW)
(ARE SUCH QUESTIONS MUCH ON YOUR MIND)
(WHAT ANSWER WOULD PLEASE YOU MOST)
(WHAT DO YOU THINK) (WHAT COMES TO YOUR MIND WHEN YOU ASK THAT)
(HAVE YOU ASKED SUCH QUESTION BEFORE)
(HAVE YOU ASKED ANYONE ELSE)))
(BECAUSE ((0) (IS THAT THE REAL REASON) (DON'T ANY OTHER REASONS
COME TO MIND) (DOES THAT REASON SEEM TO EXPLAIN ANYTHING ELSE)
(WHAT OTHER REASONS MIGHT THERE BE)))
(WHY ((0 WHY DON'T I 0) (DO YOU BELIEVE I DON'T 5) (PERHAPS I
WILL 5 IN GOOD TIME) (SHOULD YOU 5 YOURSELF) (YOU WANT ME TO 5)
...
...
((0 WHY DON'T YOU 0) (DO YOU THINK YOU SHOULD BE ABLE TO 5)
(DO YOU WANT TO BE ABLE TO 5) (DO YOU BELIEVE THIS WILL HELP YOU
TO 5) (HAVE YOU ANY IDEA WHY YOU CAN'T 5) (=WHAT))
(=WHAT))
(EVERYONE 2 ((0 (+ EVERYONE EVERYBODY NOBODY NOONE) 0)
(REALLY, 2) (SURELY NOT 2) (CAN YOU THINK OF
ANYONE IN PARTICULAR) (WHO, FOR EXAMPLE) (YOU ARE THINKING OF
A VERY SPECIAL PERSON)
(WHO, MAY I ASK) (SOMEONE SPECIAL PERHAPS)
(YOU HAVE A PARTICULAR PERSON IN MIND, DON'T YOU) (WHO DO YOU
THINK YOU'RE TALKING ABOUT)))
(EVERYBODY 2 (+ EVERYONE))
(NOONE 2 (=EVERYONE))
(NONE 2 (=EVERYONE))
(ALWAYS 1 ((0) (CAN YOU THINK OF A SPECIFIC EXAMPLE) (WHEN)
(WHAT INCIDENT ARE YOU THINKING OF) (REALLY, ALWAYS)))
(LIKE 10 ((0 (+AM IS ARE WAS) 0 LIKE 0) (=DIT))
((0) (NEWKEY)))
(DIT ((0) (IN WHAT WAY) (WHAT RESEMBLANCE DO YOU SEE)
(WHAT DOES THAT SIMILARITY SUGGEST TO YOU)
(WHAT OTHER CONNECTIONS DO YOU SEE)
(WHAT DO YOU SUPPOSE THAT RESEMBLANCE MEANS)
(WHAT IS THE CONNECTION, DO YOU SUPPOSE)
(DON'T THERE REALLY BE SOME CONNECTION)
...
...)

The technique consists of translating the code for the letter "O" to the code for the numeral 0 whenever it is encountered in the input character string. If the string consists only of items such as numbers and names and it is necessary to sort alphabetically on names, the occurrence of an alphabetic character within a name field is used to cause the code for zero to be retranslated to the code for the letter "O" by a rescans of the characters in the name field.

If no sorting is required, the retranslation can be avoided, provided that delimiters such as `FORMAT` or `GO TO` are spelled with zero within the recognizer segment of a translator. It is also necessary to redefine identifier as

$$\langle \text{identifier} \rangle ::= \langle \text{letter} \rangle \mid \langle \text{identifier} \rangle \langle \text{letter} \rangle \mid \langle \text{identifier} \rangle \langle \text{digit} \rangle \mid \langle 0 \rangle \langle \text{identifier} \rangle$$

where it is understood that the letter "O" is removed from the standard definition of letter as in ALGOL 60. The redefinition permits the inclusion of identifiers such as ODD or OOPS but prevents the use of an identifier consisting only of the repeated mark O.

This technique requires consistency of use and might result in chaos in a warehousing operation in which the letter "O" is used in parts labels with check digits.

L. RICHARD TURNER
NASA Lewis Research Center
Cleveland, Ohio

Dear Editor:

I would like to comment on Mr. Dijkstra's solution [Solution of a problem in concurrent programming control. *Comm ACM* 8 (Sept. 1965), 569] to a messy problem that is hardly academic. We are using it now on a multiple computer complex.

When there are only two computers, the algorithm may be simplified to the following:

```

Boolean array  $b(0; 1)$  integer  $k, i, j$ ,
comment This is the program for computer  $i$ , which may be
either 0 or 1, computer  $j \neq i$  is the other one, 1 or 0;
C0:  $b(i) := \text{false}$ ;
C1: if  $k \neq i$  then begin
C2: if not  $b(j)$  then go to C2;
    else  $k := i$ ; go to C1 end;
    else critical section;
     $b(i) := \text{true}$ ;
    remainder of program;
    go to C0;
end

```

Mr. Dijkstra has come up with a clever solution to a really practical problem.

HARRIS HYMAN
Munitype
New York, New York

V15#5
May '69

PL/I AS A TOOL FOR SYSTEM PROGRAMMING

by F. J. Corbató

My vantage point is that of a system designer-implementer concerned with the over-all system performance and the degree that the system reaches the goals that it was designed for. This gives me a little more detachment from the issue of whether the language is just right or not. For that reason some of my remarks will not be completely unequivocal but rather will be shaded.

The basis of the PL/I experience that I wish to talk about is mostly in the Multics system, which is being done as a cooperative project by the Bell Laboratories, the General Electric Company, and Project MAC of MIT using the GE-645 computer which is derived from the GE-635. However, I am not giving an official Multics view but rather only my own opinion as a member of the design team. In fact, it's a preliminary view because it is still early to be certain that we have analyzed exactly what is happening. Further, one has to be cautious in forming final judgments on a language, even though it is already a de facto standard, since there still is a need for a great deal of diversity in the computing field so that different techniques can be evaluated.

multics plans

To understand the context in which our systems programming was done, I first have to give you a brief review of what the Multics project's goals are. A set of papers is in the 1965 Fall Joint Computer Conference *Proceedings* if you wish to read more detail. Briefly, we are trying to create a computer service utility. In particular, we want continuous operation of pools of identical units. We want to combine in a single complex the goals of interactive time-sharing and noninteractive batch processing. We want to combine the goals of remote and local use in one system. The system

five years
with a temporary compiler

programming problem is to develop a framework which multiplexes a large amount of equipment simultaneously and yet also allows controlled interaction and sharing between users working in concert on various problems in real time.

In short, it's a fairly ambitious project, not because of any single idea but because we're trying to tie together many ideas at once. It was our judgment that we required new hardware to meet these goals squarely and that, of course, meant that we had to write almost all of our software for ourselves, including, it turned out, even the assembler. We were able to borrow a little, but not as much as we had hoped. Thus the project began basically at a research and development level, where flexibility is needed. We wanted a small team of people, because the hardest thing to do when you're groping in an unknown area is to coordinate people. We felt strongly that we had to have maximum flexibility in our implementation. (Continued on page 73)



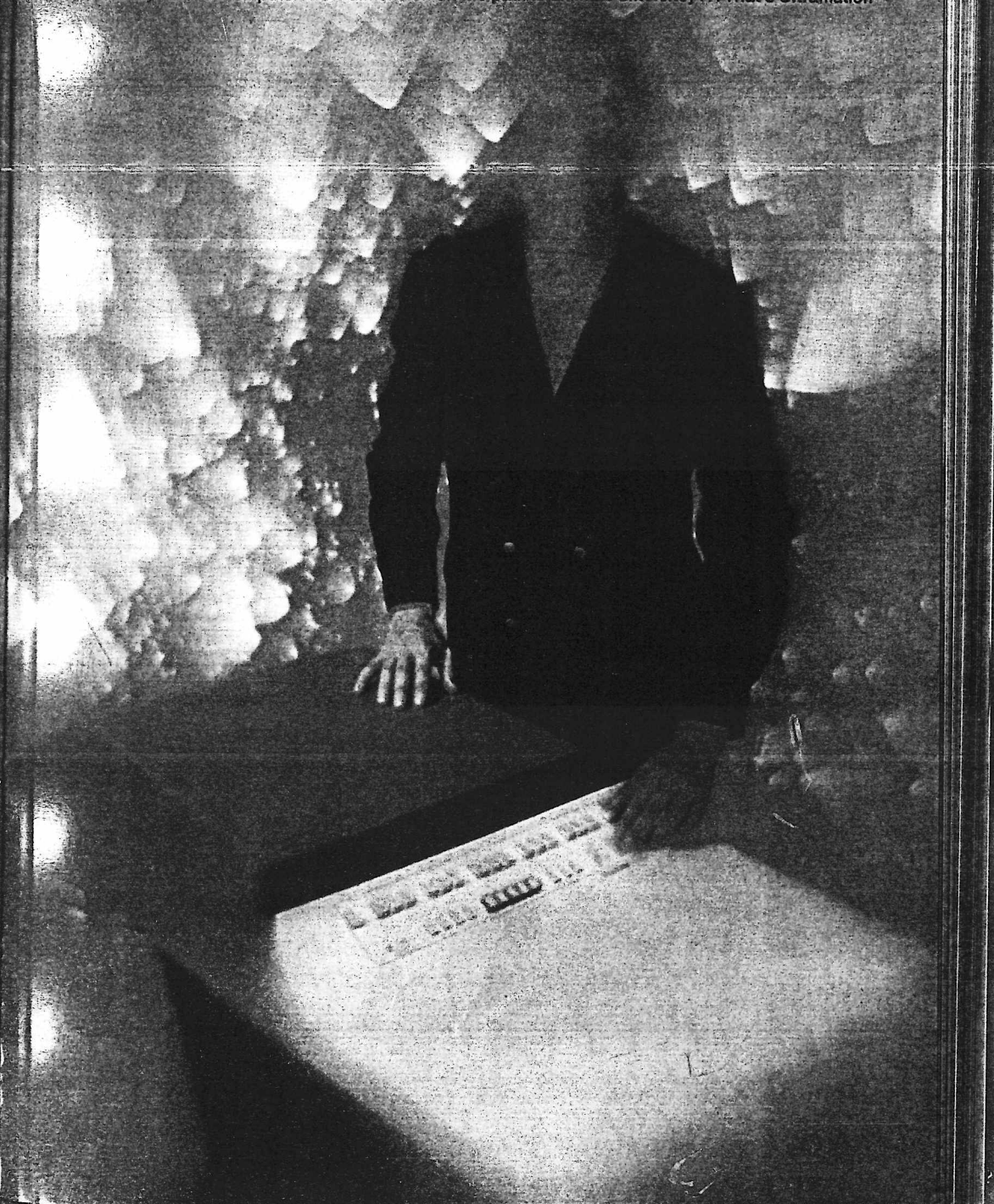
Dr. Corbató is professor of electrical engineering at MIT and widely noted for his pioneering work in development of multi-access systems, including Project MAC. He has written several books and many articles and received the W. W. McDowell Award in 1966 for his work on time-sharing systems. He has a BS from Cal Tech and a PhD in physics from MIT.

This article is based on a discussion presented by the author at an Air Force PL/I Seminar at Hanscom Field, Bedford, Mass., on March 5, 1968. The seminar was organized and conducted by Logicon, Inc., 255 West Fifth Street, San Pedro, Calif.

The ideas described here were developed in the course of research supported by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research Contract Number Nonr-4102(01).

Ultramation is here.

The Honeywell 316 computer: new dimension on the path to ultimate efficiency... That's Ultramation



Move up to Ultramation. Reserve your H316s.

SPECIFICATIONS

TYPE — Stored program, single address, 16-bit, binary, parallel, general-purpose, two's complement.

MEMORY — Type: Four wire, coincident current, magnetic core

Size: 4,096 words, expandable in same increments to 16,384

Cycle time: 1.6 μ secs

Add time: 3.2 μ secs

INSTRUCTION COMPLEMENT — 72

REGISTER COMPLEMENT (all 16-bit) —

A-Accumulator, B-Accumulator Extension, P-Program Counter, M-Memory Information, Y-Memory Address, X-Index Register

POWER FAILURE INTERRUPT — Power failure causes an interrupt through a unique location in memory to store contents of registers in memory.

INDEXING — By hardware index register, adds no time to instruction execution. Index register addressable directly and/or as memory location Zero.

SENSE SWITCHES — Four switches on control panel are capable of being tested by programmed instruction.

INPUT/OUTPUT — Word parallel. Programmed via A Register, or, optionally,

automatic to/from memory under control of Direct Multiplex Control Unit.

I/O CHANNELS — Up to 20 individually addressable channels or interfaces on the I/O bus. Multiple devices per interface or channel.

PRIORITY INTERRUPT — Basic machine has capability for up to 16 individually maskable priority interrupts. 48 additional interrupt lines optionally available.

TEMPERATURE — Room ambient.

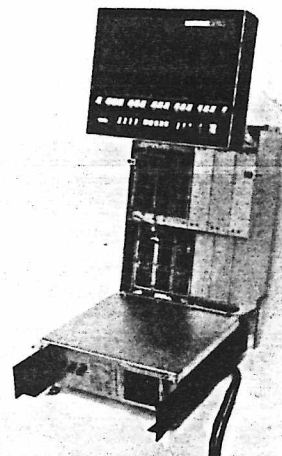
HUMIDITY — 0-90% relative humidity with no condensation.

POWER — 475 watts at 115 volts AC $\pm 10\%$, 60 cycle, ± 2 cycles, single phase. Input current, 5.5 Amps.

CONFIGURATION — The basic 316 is supplied as a rack-mountable unit complete with power supply and control panel. This unit requires 14 inches of panel height in a 19" rack. Table-top and pedestal configurations also available.

DIMENSIONS — 17.88" wide x 24.5" deep x 14" high. This unit can contain the central processor, 16K of memory, real-time clock, high speed arithmetic, Teletype interface and logic for several other internal options and/or device interface.

WEIGHT — Approximately 150 pounds.



For easy maintenance purposes, access to the unit is implemented by a front pull-out and tilt mechanism.

Honeywell
AUTOMATION

PLEASE RESERVE _____ H316(s) AND
CONFIRM APPROXIMATE DELIVERY DATE.

- ☐ Send more information by return mail. Hold this delivery date for me for 15 days so I can make a final decision and get my P.O. to you.
- ☐ Don't reserve an H316 for me yet. I need more facts. Send me your H316 brochure.
- ☐ I think I need a more powerful computer. Send me your DDP-516 brochure.

NAME _____

TITLE _____

COMPANY _____

STREET _____

CITY _____

STATE _____

ZIP _____

The application I have in mind is:

Attach this coupon to your letterhead and mail to: Honeywell, Computer Control Division, Dept. 20, Old Connecticut Path, Framingham, Mass. 01701.

CIRCLE 35 ON READER CARD

...give you a little bit of the scale of the project. I will discuss briefly the implementation. The project began in earnest in the fall of 1964 and should lead to a usable system by this fall. This means it will take approximately five years to create a useful system. That's a long time, and I think one has to appreciate the investment of effort that goes into such a venture. If you spend five years developing something you probably try to exploit it for a period of time greater than that; thus there is clearly an underlying goal here of wanting to see the project evolve as conditions change.

The system itself is described quite tersely at the level suitable for a senior system programmer in about 4000 single-space typewritten pages in the Multics System-Programmers' Manual. The system in final form seems to project out to about 600 to 800 modules of maybe four pages of source code each on the average, or, in other words, about 3000 pages of source code. The amount of system program that's in machine code is less than 5% at the object code level; it would be even less except that the compiler did not come along early enough, so some things had to be written in machine code right away. The system projects out to about half million 36-bit words which, loosely speaking, is the supervisor program. In operation, most of it, of course, pages out and is not resident in core, but it must exist and it is exclusive of all the languages and facilities of that sort, such as COBOL, FORTRAN, and even PL/I itself. The manpower to create the system has varied roughly in a linearly increasing way from zero to approximately 50 people over a five-year period.

compiler advantages

The next question I want to address myself to is why one would use a compiler at all to do system programming. (I'll take the next question of why PL/I in particular, but first: why a compiler?) First, there is the ability to describe programs briefly and lucidly. Clearly, one can obfuscate one's ideas with a compiler language but it's harder. To some extent one is talking about what one wants rather than how one wants to do it.

The trouble with machine code, of course, is that when you look at a random section of machine code you don't know what properties of the instructions the programmer really wanted to exploit. On the 7094, for example, the fact that the P-bit got cleared by an instruction may or may not be germane to what the program is trying to accomplish. With a compiler language, especially the later ones, one tends to describe what one wants to accomplish in terms of a goal and let the compiler work out the specific detail. This contributes to lucidity, of course. It also gives one the chance for change and redesign, because on a system as large as the one I have just described, the only sensible attitude is to assume that the system is never finished. Although the system obviously goes through phases, one is continually improving and evolving it.

To have had this experience on CTSS, our previous time-sharing system, and we know it is true. What happens is that users have expanding needs and goals as they exploit the facilities and they continually come up with wanted improvements. Certainly, the other extreme—of assuming that a computer software system is like hardware and can be designed once and for all on a one-shot basis and then left to the hands of some maintainers—I think has been shown to be a failure.

Another issue, too, in a system of the ambitiousness that we are talking about is that the software is at least three-quarters of the design work and yet it usually doesn't get started until the hardware is already firm. Thus there is a desire to speed up the implementation effort and using a compiler allows each programmer to do more per day. It's our experience that regardless of whether one is dealing with assembly language or compiler language, the number

of debugged lines of source code per day is about the same! Another point, too, is that the supervisor is the host of the user services so that the computer time spent in the supervisor is between 10% in some well-worked-out systems to maybe an extreme of 50% of the total time. Thus, if the compiler isn't generating the most efficient code, it isn't a disaster. In other words, one is dealing with code that isn't being exercised *all* the time. It has to exist, it has to be right, but there is room for some clumsiness. Further, if the system is well designed, the production job will run efficiently and the supervisor will remain out of the picture.

Finally, there is the issue of technical management of programming projects: the problem of trying to maintain a system in the face of personnel turnover and in the face of varying standards of documentation. Personnel turnover is expected on a five-year project. (We didn't think it was a five-year project to begin with; we estimated about half the time.) One has to assume in most organizations somewhere between 10% and 20% turnover per year even if everybody is relatively happy. People get married, husbands are transferred, and for a variety of personal reasons, people must leave, carrying with them key know-how. Training a new person involves a minimum period of six to nine months, even starting with good people, especially if you're faced with a system which has 4000 pages of description in it. You don't casually sit down and read that, even in a weekend. In fact, it's fair to say that the system is large enough that no single person can remain abreast of all parts at once. Thus there is a reasonable case for a compiler in developing large systems.

which compiler?

So the question was: What compiler to use when developing Multics? We chose PL/I. The reasons go somewhat like this. One of the key reasons that we picked the language was the fact that the object code is modular, that is, one can compile each subsection of the final program separately, clean up the syntax, and test it on an individual basis. This latter point seems obvious, perhaps, because object code modularity is in several languages, like JOVIAL, FORTRAN, or MAD, but it wasn't in the ALGOL implementation available and it blocked us from considering it.

The second reason for picking PL/I was the richness of the constructs, especially the data structures and data types which we considered to be very powerful and important features. We had a task on our hands with fairly strong requirements and with unknown difficulty. We viewed the richness as a mixed blessing, however, because we certainly were a little wary of the possible consequences. But it certainly seemed the right direction to start and maybe to err on and to cut back. As I'll get to later, it was a little too rich.

Another reason for choosing PL/I was that it was roughly machine independent. Our object in doing the system has not been to compete with normal manufacturing. Instead, our object has been to explore the frontier and see how to put together effectively a system that reaches and satisfies the goals that were set out. We are trying to find out the key design ideas and communicate these to others regardless of what system they are familiar with. Hence, a language that gets above the specific details of the hardware is certainly desirable, and PL/I does a very effective job of that. In other words, it forces one to design, not to fiddle with code. And this has turned out to be one of its strong points.

Another reason that we considered PL/I was that we thought the language would have wide support. To date it has had the support of one major manufacturer. And the final key reason for PL/I was that two persons associated with the project, specifically Doug McIlroy and Robert Morris at Bell Labs, offered to make a subset of it work. In addition, a follow-on contract with a vendor was arranged for a more polished version of the compiler. That is basically why we chose PL/I. We certainly debated, somewhat cas-

nally, other choices but these were the essential reasons why we picked the language.

The subset that was implemented initially as a deliberately quick-and-dirty job was called EPL for Early PL I. Its design characteristics went briefly as follows: It had no I/O; after all, this is a system programming language and we use the system subroutines. It had no macros except the INCLUDE macro, which worked in very smoothly with the time-sharing system, CTSS, that we were using. It had no PICTURE attributes or things of that sort which represented the control influence, except for structures, of course. It had no multi-tasking; we found the PL I specification of this to be defective in the sense that it wasn't thought through well enough, and we certainly didn't need it for a system programming language. It had various minor restrictions like requiring structure names to be fully qualified. No complex arithmetic, no controlled storage—you can simulate that easily—and, more importantly, no attributes such as `DEFERRED`, `ADDRESSABLE`, `ABNORMAL`, `NORMAL`, `USES`, or `SETS`—those things which allow the compiler to do an optimum job of compiling the code with advice from the programmer; these are sophisticated and tricky attributes, incidentally—but the reason they're not there is that the compiler didn't intend to optimize anyway, so it would have ignored advice.

To emphasize the positive, the things that EPL did have were ON-conditions and signals; it did have recursive programming. In fact, the system doesn't allow any other kind easily. It did have based storage and pointer variables, and it had `SAVE` and `FREE` for storage management. It had structures, as I've mentioned, it had block structures, and it had saving strings, which we regret to some extent because of implementation difficulties. In other words, it was a pretty potent subset from the point of view of language facilities.

not so temporary

The implementation, as I said earlier, was deliberately a quick-and-dirty job. It was expected to be merely a temporary tool to be soon replaced by a polished compiler from the vendor. The team consisted of McIlroy and Morris and two to four helpers. I am going to give a detailed and candid account of the events surrounding the EPL implementation because the nature of the events plus the very high qualifications of the people involved together with the difficulties encountered indicates that PL I has intrinsic implementation problems which we did not properly anticipate. The original optimistic estimate for making EPL work was that it was only going to take about six months. In spite of the dedication of the people involved, it took over 15 months to get a compiler that was barely usable. Further a lot of work went into upgrading it in the next 18 months, since the polished compiler of the vendor never materialized and the upgrading process had to be continued, notably first by Jim Gimpel and later by Barry Wolmen. Moreover, the EPL effort—like a gruelling relay race—wore out nearly everyone who worked on it. But to everyone's credit, the compiler works and is useful.

The language that was used to implement EPL was TMC, short for "transmogifier," which is a language system developed elsewhere by Bob McClure. It's a clever, interpretive system specifically designed for experimental language writing or syntax analysis. However, it is not easy to learn and use and, therefore, it is hard to pick up the work of somebody else written in the language.

The EPL translator was initially designed as two passes, the first one being principally a syntax analyzer and the second one basically a macro expander. The output of the

second pass in turn led into an assembler which handled the specific formatting for the machine. Later a third pass was added intermediate between the first two in an attempt to optimize the object code.

The quick-and-dirtiness came through when the original language subset specs had only a single diagnostic, namely, error. That has been expanded so that maybe now there are half a dozen, but the only help you get is that the message appears in the neighborhood of the statement that caused the trouble. The compile rate, which was never a major issue, turned out to be a few statements per second. It has been improved a little with time, but more critically the object code that is generated has improved to a respectable 10 instructions per executable statement. (There's obviously a large variance attached to these figures.)

The environment that the EPL compiler had to fit into is significant. First of all, we had adopted as a machine standard the full ASCII character set of 95 graphics plus control characters, so one of our first projects was trying to map a relationship with EBCDIC—the IBM standard.

We also intended to use the language in a machine with program segmentation hardware in which programs can refer to other sections of programs by name. Fortunately, we could use the \$ sign as a delimiter to allow us to have two-component names. We also expected the compiler to generate pure procedure code which was capable of being shared by several users each with their own data section who might be simultaneously trying to execute the same procedure. We also wanted to establish as a normal standard, although not a required one, the use of recursive procedures by means of a stack for the call, save, and return sequence, linkage information, and automatic temporary storage. We also wanted to allow the machine to have a feature which we've called "dynamic loading" in the sense that an entire program isn't loaded per se; the first procedure is started and, as it calls on other procedures, these procedures in turn are automatically fetched by the supervisor on an as-needed basis rather than on a pre-request basis. This, of course, is in conflict with any language which allows storage to be pre-declared by the INITIAL specification within any possible module that is ever used by the program. (This problem also comes up in FORTRAN.)

We also had a feature in the machine which we call segment addressing that allows one to talk about a data segment without having to read it in through input/output; rather, one merely references it and the supervisor gets it for one through the file system. In other words, we were trying to design a host system capable of supporting software constructs which make it easier for people to write software subsystems.

environmental problems

In this rather sophisticated environment, one of the problems was that much of the time was spent finishing the design of the compiler so as to implement the mating of the language constructs with the environment. The things that caused trouble were the SIGNAL and ON conditions, which are relatively tricky ideas and which clash head on with faults and interrupts. The call, save, and return conventions had to be mated into the standards of the system. Problems of non-local GO TO's and the releasing of temporary storage which has been invoked had to be licked. Most of these problems are implications of the language if one thinks it through, for the language has a lot of assumptions in it about what kind of an environment it is going to be in.

There are also little subtleties, like when you're talking about strings of characters and operators, what is the role of control characters, i.e., codes without graphic representation such as backspace, when encountered in strings. There are also obvious difficulties in that the language doesn't discuss any protection mechanisms, a feature that every system must have to implement a supervisor-user relationship.

Thus there needed to be some additional modifications made to the compiler to make that work out.

And then there are strategy problems within the implementation, such as how you're going to implement internal blocks and internal functions. These also took some time to work out and were one of the principal reasons why the compiler implementation was slow going. Further, it was done simultaneously and in parallel with the system design. I would say with hindsight that we didn't put enough effort into trying to coordinate the two. The reason we did not was that to a first approximation we felt that the language implementation was decoupleable from the main project. That was a useful thing in the early days, but as we came home toward the finish line in the design, it began to haunt us that we hadn't worked out some of these interface ideas more carefully, and we had to pay the price of redesign in various parts.

One preliminary conclusion we draw from the above experience is that PL I went too far in specifying the exact environment. There are a lot of ideas that should be subroutines and not part of the language. I don't mean they shouldn't be thought through, but to think them through is the same as putting them in the syntax of the compiler. In particular, things like SIGNAL and ON conditions could indeed be implemented as subroutine calls and be part of the environment of the host system. I don't think they belong in the language per se, although if one makes the language embrace a standard subroutine library, then I, of course, agree.

I'll say very little about the vendor's compiler. They estimated it would take 12 to 18 months. After approximately 24 months, we stopped expecting anything useful to appear. One of the principal reasons they failed was that there was a gross underestimation of the work, by a factor of three to five, and it was impossible to mount a larger effort by the time the underestimation became evident. Thus, the pioneering EPL has become the standard system-programming compiler.

Let me next talk about the use we made of the PL/I language. A strong point, we felt, is the ability to use long names which were more descriptive. People still get cryptic, but they're not nearly as cryptic as they were. The full ASCII character set is a strong point because we wanted to deal with a well-engineered human interface. The structures and the data types, as I mentioned earlier, we consider to be one of the strongest assets (this perhaps comes as no surprise to COBOL users but this feature is very important when you're trying to design data bases). The POINTER variable and based storage concept, along with ALLOCATE and FREE, have been pivotal and crucial and have been used extensively. Some of the features like SIGNAL and ON conditions, which have cost us a lot of grief, at least in principle, have been very graceful ways of smoothly and uniformly handling the overflow conditions and the like, which suddenly tie you down into the guts of the supervisor. In previous systems we have always had the quandary of how to allow the user to supply his own condition handlers in a convenient way. We're not sure that the price isn't perhaps too high, but the mechanism does look good.

Another point about PL/I, that is perhaps obvious, is that the conditional statements that are straight out of ALGOL are very valuable.

reduced errors

Over-all, the general result that we got from using PL/I was a rather small number of programming errors (after a programmer learns the ropes), in fact, a sufficiently small number that one of our major sources of residual trouble is that a lot of bugs have been caused by mismatched declarations, getting parameters in a calling sequence inverted, getting argument types in calls mixed up, all clerical errors

in which the language gives you no help and our implementation doesn't either. In fact, this is a defect in the language in the sense that the independence of the separate compilations has left a gap in the checking of types. (Sometimes programmers have used mismatched declarations for gimmicky convenience or efficiency although we have tried to avoid it because it obviously destroys machine independence.) We also found that skillful system programmers who know the machine well don't want to work in machine language because they make too many mistakes. This condition is aggravated because in modifying the machine we retrofitted a lot of involved ideas onto a somewhat ornate order code. Regardless of the reason, however, we find that programmers would rather get things done than twiddle bits.

A major effect of the use of PL/I has been that we have been able to make major strategy changes which are really vast redesigns. One example occurred in the management of the high-speed drum that did most of the paging. The program that did this was reworked, quite a while ago, when some insight developed which allowed a tremendous amount of bookkeeping to be eliminated. The amount of code that was involved dropped from 50,000 words to 10,000 words. This total rework was done in less than a month (although not completely checked out because the person wasn't working full time on it).

A second example of redesign occurred in the area of a special high-strung I/O controller which has all kinds of conventions and specialized aspects. The first cut of the control program design was a little rich; it ended up involving around 65,000 words of code. After people finished debugging it and recovered their breath, they took a closer look at it and saw that by cutting out maybe 10% of the features and changing some of the interfaces and specifications they could streamline it. Two good men working very hard did the reworking in less than two months. The two months were peak effort, but they did do it. The program basically shrunk in half, down to 30,000 words and it runs about five or ten times faster in critical places.

This kind of redesign is invaluable and in many key areas of the system has occurred several times. It gives one the mobility that one is after. It may be true for the use of nearly any compiler—I'm not trying to argue that this is exclusively a PL/I attribute—but this is the experience we're getting. Another example was a major change in the system strategy of handling "own data" sections, which we call linkage sections. We were keeping them as individual segments but we reorganized things so that they were all combined into a single segment because some of our initial design assumptions had not been correct. This reworking was done in the period of a month. The change was serial to the main line of the project development, so that it was a rather important period of time to minimize.

object code

Now, there's another side of the coin, namely, object code performance. At present the comparison to hand code seems in the vicinity of two times worse and is largely because the compiler cannot optimize very well. A lot of redundant expressions are being calculated; this is especially true with based storage and pointers where it is easy to build up fairly elaborate expressions to access a variable and then at the next occurrence repeat the calculation.

Finally, and this is perhaps the one shocking note that should be taken with some caution, we find that a typical good system programmer produces on his first try EPL-generated object code which is perhaps 5 to 10 times as poor as hand code. I think this is the main problem with PL/I, because a factor of 5 or 10 at the wrong places can sink the system. The reason for the factor of 5 or 10 seems to be principally that programmers don't always realize the mechanisms they are triggering off when they write some-

thing down. The usual pattern when one writes a program is to think of four or five ways that one can write out a part of an algorithm and to pick one of them on the basis of knowing which way works out best.

What has happened is that people are too detached. For example, if you use a 1-bit string for a Boolean variable, it turns out in our particular implementation you generate a lot more machinery than if you'd used a fixed integer. Similarly, varying strings carry a fairly stiff price tag in our present implementation (although ways are known to improve matters a little), and they must be used with caution. Occasionally, too, we've had mishaps where the machine independence works against us in the sense that a man declares an array of repeating 37 bit elements and the compiler dutifully does its straddling work boundaries mercilessly. The best we've been able to do so far is to get the compiler to at least remove the direct code listing the word "array." There may be other reasons for the factor of 5 to 10, such as the language learning time, but we do not consider them important. Other issues, such as the 10 to 1 variation in ability among programmers of similar experience (discussed in an article by Sackman, Erickson and Grant in the January 1978 *Communications*), I think can be discounted in our case. Our technical management has been firm, but we have kept track of the individual programmers so that mismatches with work assignments have been minimized.

With regard to technical measures for upgrading the system one must remember that most of the code in the supervisor doesn't matter; it's not being used most of the time so the key thing is program strategy. I don't have time to discuss here how we organize the parts of the code which are the functionally important parts, but a segmented machine pays off handsomely. Meanwhile we are learning trade-offs between the direct supervisor mechanisms. In addition we are trying to develop the lists of things to avoid in the language. It turns out to be rather hard to get people to adhere to it, but we're going. On a long-range basis GE is developing plans for an optimizing compiler, but it isn't going to help us right away. We are also studying on a preliminary basis smaller aspects of PL 1, with perhaps modifications and changes to the language so that the implementation is more genuinely potent or impotent, depending on how you look at it. That is, the user would be constrained to a language which would implement well regardless of whether he takes one choice or another. And, finally, there are some coding tricks that might have helped if we had thought of them sooner.

another problem

One of the key problems in our use of PL 1 has been that the programmer doesn't have feedback. If he had, say, a time and space estimate on each statement that he writes, given back by the compiler, and if he were in an interactive environment developing the program (i.e., he could get quick return on his compilations), he might be able to form some intuition about what he's doing. To implement such estimates is not a trivial problem, because a lot of the mechanisms that are invoked are shared, so that there needs to be a way of designating the shared mechanisms and showing why they are included.

As a last resort in improving supervisor performance we can always go to machine language on any supercritical module. But this isn't a panacea, because it is easy to be swamped if one tries to put too much in machine language and moreover one has lost mobility. (Going to machine language should be compared to parachuting out of an airplane.)

One of the big advantages that one might expect from

the use of PL 1 as a system programming language is the ability to transfer software from one operating system to another. However, if you get totally wrapped up with an operating system which is peculiar to a particular set of hardware you're trapped. You're just going to have to rework it on another machine. It is a long-range goal, as yet unachieved, to minimize this problem. We've considered the problem some because one of the obvious questions that arises when you have a system which has been largely implemented in PL 1 is: Could you put it on another machine? The answer is "Yes," although I think it's still at the level of a "technical challenge" even with similar hardware. To do it one would have to go through and modify and edit all the programs to make it come out just right. There are also some strategy changes required, probably, unless one actually built an identical machine. Nevertheless, such a task is still preferable to having to start all over, writing off as a total loss the ideas and efforts of several years; this is the alternative to working in a compiler language for system programming. I don't really see how there's going to be any progress in the field until we stop killing off our system children.

Finally, I have a few general conclusions. I think that in the language area there has been considerable swinging of the pendulum. FORTRAN was the first compiler with any widespread use and it suffered because it wasn't systematic to implement and was somewhat clumsy to use. It was, however, a practical language. ALGOL was, in a sense, a reaction, but it suffered because it left out the environment and didn't come to grips very squarely with the implementation. PL 1, in effect, is a reaction against ALGOL's not having considered the environment, but it suffers from being designed without well-formed plans for a systematic implementation. The notion of "systematic" is important because without it the cost of implementation, the speed of the compiler and the quality of the object code may be off by factors of 10 or 100. Nevertheless, I admire the PL 1 design effort and consider it valuable because it has inspired language experts to try harder; in effect, it has set as goals what is wanted. The fact that the language has not been implemented well by anyone, I consider to be an object lesson. Nevertheless, techniques for mastering the problems are being found. In addition, people are beginning to think of ways of accomplishing the same functional characteristics without the same internal problems. One of the ways is to try to minimize the language syntax and to think through more carefully what is the subroutine library.

Future languages will come and they'll be beneficial, too. But PL 1 is here now and the alternatives are still untested. Furthermore, I think it is clear that our EPL implementation is going to squeak by, and in the long run the Multics project will be ahead because of having used it rather than one of the older languages.

Now the last question, which I think is a tough one: If we had to do it all over again would we have done the same thing? I'm not totally sure of my answer; I just don't know. We certainly would have designed the language more carefully as part of the system; that was something we didn't pay enough attention to. If it was EPL or a PL 1 again we would have tried to strip it down more. With hindsight we would have modified it to some extent to make sure that it could have been implemented well. If it were another language, we would have tried to beef it up with things such as are in PL 1, and maybe modified it. Either course of action takes a lot of design time, and that's the dilemma: in effect, one wants one's cake and one wants to eat it, too. I think the decision probably hinges on whether or not one is trying to meet a deadline. I would probably use FORTRAN to meet a firm deadline. But if I'm trying to solve a problem with a future, I think I would use either PL 1 or its functional equivalent—and the choice will have to be answered in the future.

NEW DIRECTIONS IN INFORMATION DYNAMICS

Instant playback of yesterday
Better decisions for today
Improved profits for tomorrow

For Stromberg DatagraphX Circle 136 on Reader Card

DATAMATION

How long would it take us to show you the advantages of the new 315 RMC Rod Memory Computer?

About 800 billionths of a second.

Because its entire main memory is made up of thin wire rods plated with a magnetic thin film that permits cycle speeds of 800 nanoseconds (800 billionths of a second). That memory, incidentally, has a capacity of up to 240,000 digits.

Also available for this new 315 RMC — and all 315's — is a new line of faster and more efficient peripherals, such as: speedier tape drives (66 KC conversion of data from other computers and 120 KC for direct processing); a 1000 line-per-minute printer; a 250-cpm card punch; a data communications controller for "on-line" applications; built-in floating point arithmetic; new high capacity CRAM III (Card Random Access Memory) that has up to 16,000,000 characters of random access storage in each CRAM cartridge.

The 315 RMC is completely compatible with all existing 315 peripheral equipment. All 315 programs and software, including NEAT, COBOL and the recently announced program generator BEST may be run—as is.

Deliveries begin in mid-65. For more information about the versatile 315 family of computers (and, especially, the new 315 RMC), call your local NCR office. Or, write to NCR, Dayton, Ohio 45409.



NCR

BE SURE TO VISIT THE NCR PAVILION AT THE N.Y. WORLD'S FAIR

THE NATIONAL CASH REGISTER COMPANY



C. L. McCARTY, JR., Editor

A Multiuser Computation Facility for Education and Research

JACK B. DENNIS

Massachusetts Institute of Technology, Cambridge, Mass.*

Present-day computing facilities are limited in their value for scientific research by inability to interact strongly with users. The full power of a research computing instrument should be available at many terminals that give each user the ability to generate, correct and operate any procedure he wishes, either simple or complex.

Implementation is described for a small-scale multiuser computer system that permits several users to work independently with the machine, and to obtain satisfactory response using typewriter communication.

With the maturing of the art of computation we are now in possession of vastly more powerful calculating instruments in terms of speed and memory capacity. Procedures of great complexity are amenable to mechanization, and simple calculations are accomplished so fast that the start and finish cannot be resolved. It is now apparent, however, that present means of communicating with automatic computers are not compatible with these greater capabilities: too much effort must be expended to set up even the simplest calculation for automatic operation. Techniques for mechanizing complex procedures are antiquated, and do not take full advantage of the internal capabilities of present-day machines.

It is now recognized that improvement of this situation requires the ability of a machine to interact strongly with its environment—meaning physical sources of data, and equipment that is being controlled, as well as the human

This work was supported in part by the U.S. Army Signal Corps, the Air Force Office of Scientific Research, and the Office of Naval Research; in part by the National Science Foundation (Grant G-16521), the National Institutes of Health (Grant MH-04737-03), and the National Aeronautics and Space Administration (Grant NSG-496); and was sponsored in part by the Advance Research Projects Agency, Department of Defense, through Office of Naval Research Contract Nonr-4102 (01).

* Department of Electrical Engineering and Research Laboratory of Electronics.

users of the machine. The provision of computers with this ability is crucial to their effectiveness for education and creative research.

At present, available commercial machines still fall short of providing the kind of interaction ability needed for progress in scientific applications. This state of affairs evolved from the demands placed on computer manufacturers by their best customers—government and industry—for whom creative scientific research plays a relatively minor role. Present machine designs are oriented toward administrative and operational data processing.

Thus, it falls to the basic research community—educational institutions and prominent industrial laboratories—to emphasize the need for a transformation in the quality of computation facilities available for scientific work. The M.I.T. Long Range Computation Study Committee has examined this problem in detail [2] and has reached this conclusion:

The computation needs of education and research can be met best by a large, powerful computer system directly controlled from a number of remote consoles which provide rapid access to the full capabilities of the central machine, and a quick response to simple tasks requested by its users.

The subject of this report is a small-scale computer system designed to provide the qualities of access and response envisioned by the Study Committee. The background and motivation for this study are presented in Part I. Part II gives a functional description of the computer system in its initial form. The performance of this system will serve many research and teaching needs and will help in the planning of future large-scale facilities.

PART I. BACKGROUND

Character of Academic Computer Usage

To understand the character of a computation facility needed in an academic environment, the nature of computer usage in a research-oriented university must be examined.

Program Development. Perhaps the most significant observation is that the major portion of academic computer usage is devoted to program development. In education, computers are used in teaching the arts of computation—programming languages, numerical methods and other classes of procedures. Students of these principles learn most effectively by writing, testing and running their own programs. The computer also serves students as a calculator.

lating instrument for term projects and theses in which each task generally calls for the coding of new combinations of familiar and original procedures.

Research in computer technique is concerned with the design and evaluation of algorithms for specific classes of mathematical and information-processing problems, and inherently involves the preparation of suitable programs.

The applications of computation to many broad areas of creative science and engineering need programs as tools of research. New tools are continually being fashioned in the form of programs of greater power and with more convenience of use.

Program development requires considerable communication between the human user and the machine. An algorithm must be stated to the machine in an appropriate programming language. The source-language program is then converted into a sequence of machine instructions by a compiling or assembling program. The process of program development would end here, were it not for the difficulty of writing down with complete accuracy the typically rather complex sequence of steps making up a program statement. The more complex the procedure, the greater become the problems of accuracy. During the translation of the source-language program, format inconsistencies and other mistakes frequently show up which require editing of the procedure statement. Once a successful translation of the program has been accomplished, errors in the formulation of the algorithm must be traced and corrected. These steps all require communication with the machine, which is painfully slow with current machine-operating practice: a powerful and costly computer must be continually utilized. Therefore programs are generally run in batches with at least several hours, or as much as several days, required to conduct a brief "conversation" between user and machine consisting of a single statement to the computer followed by the computer's response. The staffs of computation centers compensate for this restricted communication ability by providing overelaborate responses from the machine; the user compensates through elaborations on the statement of operations to be performed. These consequences are wasteful of computer capacity and—perhaps more important—wasteful of human effort.

Experimental Computation. A frequent use of digital computation in research work is modeling the behavior of a physical system. Once a program for performing the simulation has been developed, the researcher generally wants to evaluate its performance under a variety of conditions, or to adjust parameters of the system for optimum performance. In general, convenient access and quick response are required for the power of the computer to be of greatest value to the experimenter.

Conducting Experiments with Online Computation. In many research applications the computer is used as a sophisticated data-processing device in which the data sources are experiments that are being carried out in research laboratories. Some salient examples are nuclear physics experiments with high-energy particle accelerators,

direct observation of the behavior of animal nervous systems, and the evaluation of psychophysical experiments. A number of other possibilities awaits adequate computation facilities for serious exploitation.

At present, it is generally necessary to record data during the experiment, and process it later when computer time is available. Not only does this require expensive equipment or tedious transcription, but the delay in processing and presenting results is frustrating to the experimenter. Frequently, the processed data if immediately available would enable the experimenter to change the conditions of the experiment to obtain more significant data. In many cases the computer could even control the conduct of an experiment with more accuracy and precision than the human experimenter, and with more flexibility and sophistication than he could afford the time and effort to build into special apparatus.

In some instances, small general-purpose computers are used to process such data online. The character of the processing, however, is quite limited by the capabilities of the equipment being used. Computation facilities are needed which make the capabilities of a powerful processor available for online experimental applications.

Unexploited Applications. We have examined the way in which present usage of computation in the university suffers from the inadequate access and response properties of our present computation facilities. As more complex procedures are devised for research studies, this lack will become increasingly apparent. Furthermore, there are many problems for which digital computation would be used and services that a computation facility would provide, except for the difficulty of access. But for this problem the desk calculator would be obsolete. Services such as commonly used numerical procedures, automatic algebraic manipulation, symbolic integration, the indexing and retrieval of documents, can be mechanized effectively with known techniques. These techniques are useless with present facilities, for it is too much trouble for the user to phrase his simple question to the computer.

Quality of Facility Needed

From the nature of academic use of computation, the ideal academic computation facility must possess certain basic qualities. Of greatest importance are the qualities of access and response. The facility should be continuously accessible to its users and should react quickly to requests for simple tasks of computation.

Terminal Devices. To obtain these qualities it is necessary to provide appropriate terminal devices for interaction between the user and the machine. The medium of communication should be that which provides the greatest convenience for the user. In our experience with online use of computers, a number of devices have proved particularly useful for communicating with the machine.

1. **Typewriter.** At present, our techniques for preparing procedures for automatic execution are totally dependent on character strings as the medium of communication.

The typewriter is now the cheapest and most convenient device for communicating with a machine in this medium. A typewriter requires only a very small portion of the capabilities of a computer for its operation. Available typewriters are far from ideal for this service, principally because they are overly expensive to connect to a computer and are limited in terms of the character set available to the user.

2. **Graphic Display.** The cathode-ray-tube display is a very flexible instrument for presenting information to a user in graphical form. Although more expensive than a typewriter, a display allows a visual presentation of graphs and diagrams which is easier to grasp and interpret than printed lists of numbers. A display may also be designed to permit rapid presentation of text, so that the combination of keyboard and display can substitute for a typewriter. This has the disadvantage, however, that no permanent record is made of a "conversation" between user and machine. There is a variety of possible levels of sophistication in the design of display systems with a corresponding variation in cost. The kind of display system that is most suitable for general use, and offers a good compromise with expense, is a matter of considerable debate.

3. **Light Pen.** A light pen is a pen-shaped device that responds when a point within its small field of view is intensified on a cathode-ray display. In conjunction with a display, a light pen gives the user a convenient means of entering two-dimensional relationships among quantities or objects, and a handy way of selecting a particular member of a large set of functions or objects.

4. **Buttons and Knobs.** Buttons and knobs have been found to be valuable adjuncts to a display/light pen combination. Simply by pressing a button, the user may quickly select the mode in which the light pen is being used, or choose some operation to be performed on information that is being displayed. Knobs have been arranged to give manual control over several parameters of a problem so that their effect can be directly observed in the displayed results.

Form of Information-Processing Capacity. In view of experience with the means of machine-user interaction described above, it is desirable to make these media the primary user terminal devices of a research computation facility. The form to be taken by the processing capacity of the facility is the next question to be studied.

Two major alternatives for serving a large number of users with limited resources are apparent: the processing capacity could be realized as a single central system serving many users simultaneously, or the processing capacity could be split into many relatively small units, each serving one user at a time. The use of many small machines is not a satisfactory solution, as it is possible to provide superior service from a central computer system for much less expense through the techniques of time sharing.

The size of machine that an individual can afford immediately places a limit on the complexity of procedure that he can mechanize and the amount of data that can be

processed. On the other hand, an individual user, although he needs the access and response qualities that we have emphasized, cannot in general effectively utilize the computation capacity of even a small computer. He needs to ponder results of a calculation to determine the most appropriate next step. Furthermore, the amount of processing capability and memory, and the quality of interaction with the machine that a user requires vary greatly in the course of time, and always afford an imprecise match to the capacity of a small machine.

Other disadvantages of the small private computer are readily apparent. It cannot be applied to complex procedures requiring large amounts of main storage. Sophisticated compilers would not be available. Internal program files are not possible without inordinate additional investment in equipment.

Properties Required of a Central Data Processor. These arguments lead to consideration of how a centralized computing system can be arranged to provide high quality of access to its users. The general layout of such a system is shown in Figure 1. From the above discussion of the nature of needed performance, certain fundamental requirements on the design of the system are evident.

1. It must be possible for a user of the system to write, edit, test and run an arbitrary program in any available language system, through appropriate use of his terminal or console.
2. The operation of one user's program must not be allowed to affect the computation performed by another user's program, except in the time required for completion.
3. The system must be able to execute simple procedures with convenience and efficiency.
4. The central machine must be capable of providing user-program interaction of a quality suitable for the terminal devices, especially the typewriter and display oscilloscope mentioned above.

To attain the quality of access envisioned here, the capability of the central machine must be such that executing responses to simple queries from all user terminals requires only a fraction of the machine's processing capacity. It is also evident that a large computation facility with the above-mentioned properties is only feasible if the system has a large enough body of users to make the average computation load comparable to the capability of the machine.

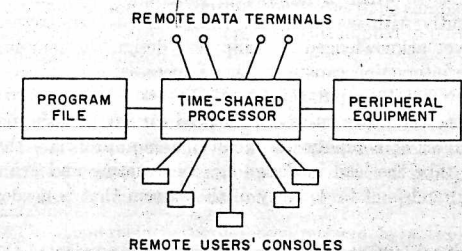


Fig. 1. General layout of a large time-shared computer

Realization of a Time-Shared Computer System

The general configuration of a system having these qualities is shown in Figure 1. A powerful processing and memory element is coupled to many remote consoles for communication with users, and a number of data terminals for connection to experiments. A large auxiliary memory, labeled "program file" in the figure, holds all programs so that they may be called up by a user from his console whenever needed. Peripheral equipment permits handling of large printing jobs, loading data from special media, graph plotting, photographing of displays and other miscellaneous functions.

The central processor of the system must be able to shift attention among independent tasks from many users. For this to be successful, the system must have certain properties. It must be impossible for one program to interfere with correct operation of another program except in the time required to complete a computation. Specifically, a program is prohibited from modifying the contents of storage areas not allocated to it, or operating input or output devices not assigned. Ease of preserving all machine conditions pertinent to the operation of a program is essential for quickly shifting from one program to another.

In terms of storage requirements, the process of shifting can be handled in several ways. One way is to have a main memory large enough to accommodate simultaneously all programs that are active at a given time. In this scheme one program may occupy different locations in the memory on the successive times that it becomes active. Hence means must be provided to allow a program to operate correctly from any area of memory. Alternatively, active programs may be preserved on an auxiliary-storage device and brought into the main memory to operate in succession. It appears that in a large time-shared computer system, the best choice will lie somewhere between these extremes.

Available computing equipment is not now oriented toward time-shared operation in the sense with which we are concerned here. Machines have been designed to share processor capability among standard input/output equipment, and therefore include the facility to shift the processor's attention among subroutines associated with each in/out device. No attention, however, is paid to protecting one routine from the actions of another, or to saving all conditions pertinent to a program's operation. Although several manufacturers have recently announced computing systems in which a number of programs may run concurrently with complete independence, no manufacturer has yet acknowledged in equipment design the man-machine interaction capability that is necessary for effective experimental computation.¹

Since the universities have a need for this transformed character of computation facility, it is appropriate that they take the lead in designing, constructing and evaluating models of the quality of the system that is needed.

¹ Several large-scale systems have since been announced that go far toward achieving our objectives.

This reasoning has stimulated two projects at M.I.T. aimed at demonstrating the feasibility and desirability of a time-shared multiuser computation facility. One study is being pursued by the M.I.T. Computation Center, with an IBM 7090 computer used as the central processor [1, 4]. The study described here was made possible by the donation of a PDP-1 computer to the Department of Electrical Engineering, M.I.T., by the Digital Equipment Corporation of Maynard, Massachusetts.²

PART II. THE MULTIUSER SYSTEM

Characteristics of the Basic Computer

The PDP-1 digital computer used in this study is a small-to-medium scale machine, with binary arithmetic, parallel operation, and a magnetic-core memory having a 5- μ sec cycle time. Its system organization is shown in Figure 2. The arithmetic element consists of the accumulator and memory buffer register, with the in/out register included for shifting, multiplication and division. The machine performs arithmetic with one's-complement binary numbers, Boolean operations on 18-bit words and a variety of shift operations. The basic memory element contains 4096 words of 18 bits which are directly and indirectly addressable. The memory is expandable to 65,000 words. Random access to the expanded memory is obtained through indirect addressing in an *extend mode* selected by the program.

Standard input/output equipment for the PDP-1 includes a typewriter through which the user may type information to an operating program, and the program may print data and comments in reply. A photoelectric tape reader operating at 400 characters per second is the normal means of placing programs and data in the machine. A paper tape punch may be used to preserve results and to prepare machine language forms of programs written in a symbolic source language. A display system permits graphical presentation of results as plots on a cathode-ray tube. A light pen can be used with the display as a versatile graphical input to the machine.

The basic machine includes a sequence break feature designed to allow efficient programming for the various

² Since this writing, an identical system has been set up for Project MAC at M. I. T. with the objective of developing the concept of an "information processing utility" and exploiting its use for scientific research. Similar activities are in progress at a number of other universities and research institutions.

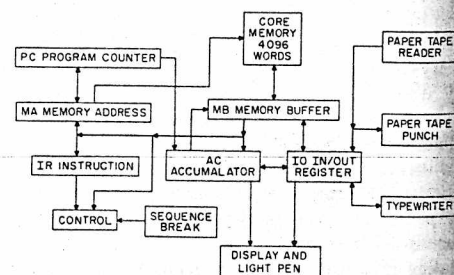


FIG. 2. Organization of the PDP-1 computer

in/out devices. When an input device is ready to transmit a unit of data, it interrupts the operation of a main program and gives control to a service routine that accepts the data, then returns control to the main program. A similar sequence occurs when an output device becomes ready to accept the next unit of data.

Normal Use of the Machine

With the standard machine the process of program development usually proceeds in the following manner. Source language programs are prepared in an assembly language in the form of punched paper tapes by using offline tape-punching typewriters. The source program tape is then processed by the assembly program which punches an object program in the form of a binary-coded paper tape. The assembler also produces a paper tape containing the values assigned to the various symbolic addresses and parameters used in the source program. The process of assembly generally uncovers a number of clerical errors in the coding of the source program. The user may decide to edit the source program offline and repeat the assembly, or he may correct the machine-language object program if the errors were simple and few. In any case, all but the simplest programs are certain to contain errors in their coding which do not show up during assembly, and must be tracked down by running the program and examining intermediate results.

A program known as DDT (for debugging programs) has been prepared to help the user in his detective work. The user communicates with DDT through the online typewriter. By typing requests to DDT, he may examine and modify the contents of any register of his program or data, or search memory for occurrences of a particular instruction or references to a particular location. Break points may be inserted into the program so that computation stops when a specified step in a program is reached, thereby allowing examination of intermediate results.

An editing program is also available for correcting source language programs through the online typewriter. Lines may be inserted, deleted or replaced through simple commands interpreted by the editor.

Multiuser Operation of the Machine

The objective in our design of a multiuser system around a PDP-1 computer has been to meet the general goals for an academic computation facility outlined in Part I of this paper, in the form of a relatively small installation. In particular, we have tried to retain as far as possible, with available funds, the ability of users to communicate with their programs through typewriter, display and light pen. The modest size of the system allows us to adopt the philosophy of placing the qualities of interaction ability and response time for simple tasks ahead of attaining maximum efficiency of execution for long programs. We have strived to permit each user of the system to specify and operate any procedure that is feasible for the basic computer. And we have designed the system so that the computation performed for one user cannot interfere with the accurate performance of other users' tasks.

Because of the high expense of display equipment and the limited capability of the PDP-1 in maintaining many simultaneous displays, it was necessary to emphasize the typewriter as the principal means of communication between user and machine. The display and light pen are incorporated in the system and are assigned to one user at a time.

In our machine, only a single core memory unit of 4096 registers is available, and it was not deemed practical to relocate programs within this memory. Consequently, only one user's program at a time may occupy core memory, and multiuser operation must be accomplished by running programs in time sequence. With several programs operating, each one is placed in the core memory and run for short time intervals while the other programs are preserved in auxiliary storage. A unique magnetic-drum storage serves this function in the present system. It has 22 fields of 4096 words each. One field can be read into the core memory while the former data in the core memory are simultaneously written on a second field. This swap takes place in just 33msec for a complete field of information. Clearly the speed of the swapping operation is crucial to the performance of the system, and the above-mentioned time presses the limit imposed by the cycle time of the core memory.

At present, the system includes three consoles and one display, although provision has been made for expansion to seven consoles and two display tubes. In line with our objective of providing for interaction with laboratory experiments, the system includes a facility for attaching experimental equipment to be operated by the system.

Sequence of Program Operation

To supervise the operation of users' programs within the system, a section of core memory is reserved for an executive program. One function of the executive routine is to determine the sequence of operation of users' programs. As it was necessary to keep the executive routine small to make as much memory as possible available to users, a round-robin scheme was adopted. Each user is considered in turn, and his program is brought into memory and run if it has a task to perform. Operation of a program is terminated when a preset time limit is exceeded. The program is then returned to the drum and the next user's program placed in operation. The time limit used is approximately 120msec. Operation of a program is also ended if it has no further computation to perform, that is, if it is waiting for input/output operations to be completed, has reached a halt instruction or has attempted to perform an illegal instruction.³

Handling of Console Conversations

In view of the time-shared mode of operation outlined above, it is not possible for a program to exercise direct control of a console typewriter and operate it at full speed. In typical typing functions performed by the PDP-1, the

³ The round-robin queue has been replaced by a two-level queue. Computations start in the top level and fall to the second level if they exceed a preset duration.

machine spends most of the time waiting for the typewriter to complete an operation so that the next character may be transmitted. So that this wasted time may be utilized by other functions of the system, and so that a user's program may run a typewriter continuously at full speed, a set of buffer registers is included within the executive routine for each console. When a user's program gives a type-out instruction, the executive routine enters the character to be typed in the appropriate buffer table. The executive routine periodically interrogates the console typewriters and transmits characters to them from the buffer tables as they are able to accept them. For typewriter input, the executive routine enters characters in the appropriate buffer table as they are typed, and transmits one character to a user's program each time it issues a type-in instruction. If a type-out instruction is given by a user's program and the buffer table is full, or a type-in instruction is given when the buffer is empty, the program is dismissed until the buffer status changes so that it may continue.

As we have explained, normal use of the typewriter in working with the computer is in conversations conducted by a user in brief phrases with his program. Frequently, a single character typed by the user calls for an action by his program. For example, in using the program-testing routine DDT, typing a backspace causes the contents of the next register in sequence to be printed; typing a slash prints the contents of the register addressed by the instruction last printed. The intimacy of interaction and flexibility of language syntax provided by the possibility of a program reacting to each character typed by the user is quite significant. Its realization is worth considerable design attention and some additional system expense.

In terms of the multiuser system, the character-by-character concept requires that a user's program be placed in the round-robin queue as soon as any character has been typed into its executive routine buffer.

Console Control Panel

Providing the user with a console consisting just of a typewriter for communicating with his program deprives him of several convenient features that are available in normal operation of the machine with the online typewriter. The standard PDP-1 is equipped with six sense switches that can be set by the user and examined by his program to determine the course of a computation. Also, the console of the machine includes a lever that will start operation of a program at an address specified by a group of switches. The most common use of this lever is to return control to a program-testing routine such as DDT after a program that is being debugged has hung up or entered a long or unending loop. At other times, the start lever may be used to initiate a particular phase of computation by one's own program. It is felt that these features are sufficiently valuable to warrant including them as a part of each console of the multiuser system. A small control panel illustrated in Figure 3 is associated with each console typewriter. The sense switches on the panel sub-

stitute for the standard sense switches while a program assigned to that console is operating. The "call lever" returns control to a program previously specified by the user to the executive routine. Each console panel also includes a "console-on" switch that a user turns on to indicate his desire to use the facility. In consequence, an administrative routine is placed at his disposal so that he may request assignment of in/out devices and obtain the services of a program loader, program editor, assembly program or debugger according to his need. Turning the switch off, then on, allows the user to communicate with the administrative routine so that assignments may be changed. Leaving the switch off indicates to the administrator that the user is through so that input/output devices and memory assigned to him become available to other users.

The "stop print" switch on each console panel inhibits printing by the typewriter so that the user may type a new command to his program. The "type" indicator light shows when type-in is permissible, that is, the executive routine buffer is not full. The "run" lamp is turned on while the program assigned to the console is in core memory.

Display System

The standard cathode-ray display system of the PDP-1 plots one point on a 1024 x 1024 grid for each display instruction executed by a program. The horizontal and vertical coordinates are given by 10-bit numbers in the accumulator and in/out register when the display instruction is given. The programmer creates a visual pattern on the display screen by plotting many individual points in rapid succession. In most cases the coordinates of the points are computed as they are displayed; that is, the pattern to be presented is described by an algorithm for generating the points rather than by a list of the points themselves. Thus the major portion of the machine's computational capacity is required to produce a single display of the flexibility and quality that are possible with the standard PDP-1.

If we are to give all users of the multiuser system access to the display with this quality, only one display presentation can be produced at a time, and the majority of the machine's capability must be made available for this purpose. This is not inconsistent with making enough capacity available to other users so that they may carry on type-

writer conversations with satisfactory response. On the other hand, the user of a display presentation generally does not need to observe the image continuously, but will be satisfied if the image can be called into view rapidly when desired. The scheme adopted for the present provides each console with a display lever with which a user may activate a display program. Pressing the lever causes the executive routine to activate display instructions in the user's program and assign a major portion of the system capacity to that program until the display lever is released. Two operators working with typical display programs can readily share the use of a single display oscilloscope with this arrangement.⁴

Program File

In the multiuser system it is important to provide a means of filing source and machine language programs on auxiliary storage for several reasons. An internal program file provides a user with the convenience of calling programs from the file by name through typing a command at a console rather than by handling paper tapes. Also there would be a traffic problem with many users attempting to use one tape reader for entering programs as they are needed. Therefore, the commonly used loading, editing, debugging and assembly programs, and important subroutines, are kept permanently on one section of the drum. Another area on the drum files programs for users as long as they are operating from a console. A user enters his own programs into the drum file through the paper tape reader at the beginning of a session of usage. Then he may retrieve them from the file as needed by typing a simple command. After editing or assembly, newly formulated programs may be entered in the library. Updated programs may be recorded on paper tape before the user leaves his console. When he is finished, his programs are deleted from the file, so that the space is available to other users. Management of the program file is the responsibility of the administrative program.

It would be desirable if users' programs could be stored indefinitely in the system so that paper tapes would be unnecessary and the read-in procedure dispensed with. Another benefit would be the possibility of locating consoles remote from the computer. Such an arrangement is not possible with the present system because the drum does not have sufficient storage capacity.

Relation to Users' Equipment

A major objective of the PDP multiuser system is to provide means of tying physical experiments into the system so that the machine may process data in real time for presentation to an experimenter, or so the progress of the experiment may be accurately monitored and controlled by a computer program.

To provide for these applications, the computer facility must have the physical means of transmitting information to and accepting information from the experimental en-

⁴Operating experience has shown this scheme to be unnecessary.

vironment, and means by which the machine may be notified that data can be accepted or are ready for transmission.

In the present multiuser system, operation of special users' equipment is accomplished through a group of in-out transfer instructions specifically reserved for the purpose.

It is undesirable to assign a particular set of instructions to one user permanently. These instruction codes would then be unavailable to other users as long as the set remained specifically allocated. Also, the set of instructions must be illegal for other users of the facility so that they cannot inadvertently interfere with a production experiment.

The scheme adopted allows each user of special equipment to use the same group of instructions for its control. The equipment activated by the instructions is determined by a set of assignment signals that are controlled by the executive routine. When a program communicating with special equipment is brought into core memory and placed in operation, the assignment signal enabling the special instructions to operate the equipment is turned on. The other assignment signals are not turned on, so that the use of the special instructions will only perform the actions desired by the active program.

Sequence-Break System

In normal operation, the sequence-break system permits a program to perform computation through the operation of a main program while simultaneously performing several input/output functions through a service routine, with no special provisions necessary in the main program. The sequence-break feature does more than provide for simultaneous operation of the in/out devices: It is a convenient means by which an external event may cause a program to switch its mode of operation. For instance, a program executing a lengthy computation may be interrupted by typed information so that its progress may be determined, or a display program may be interrupted by a response from the light pen which gives control to a tracking program. The interrupt feature is thus important for establishing strong user/machine interaction. Therefore the PDP-1 time-sharing system has been designed so that the sequence-break feature is available to each user. Of course, the executive routine must interrupt the progress of a user's program to perform its functions of handling buffer operation and the sequencing of programs. This is accomplished by an executive sequence-break channel that has priority over users' breaks.

Operating Principles

To implement the multiuser system and achieve the goals outlined above, it was necessary to augment the standard PDP-1 computer with some equipment designed specifically for this purpose. The magnetic drum that can swap information with the magnetic core memory was developed by the manufacturer of the PDP-1 specifically to facilitate time-shared operation.

Other additions were, for the most part, minor but im-

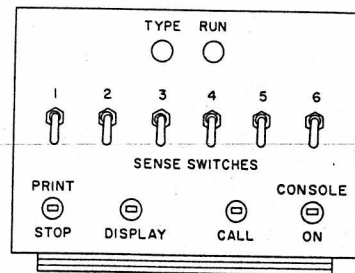


Fig. 3. Console control panel

portant additions to the logic of the standard machine for the following purposes:

1. To prevent interference among programs running for different users.
2. To allow efficient operation of the executive routine in a small amount of memory.
3. To facilitate conversations between users' programs and consoles.
4. To permit direct operation of external devices by user's programs.

The specific hardware features that have been added are the following.

1. A set of executive instructions which allows the executive routine to perform its functions.
2. An assignment register that contains a memory bound applicable to a particular user, and codes indicating what in/out devices (console, punch, reader, display and external device) are assigned to this user. It is loaded by an executive instruction.
3. A two-channel sequence-break system. The high-priority channel defines executive mode and establishes operation of the executive routine when a trap or interrupt condition is met (as described below). The low-priority channel is arranged to appear to each user as the one-channel sequence-break feature of the standard machine.
4. Control logic for up to seven typewriter consoles.
5. An alarm clock for timing operation of user's programs.
6. Selection logic to permit connection of external apparatus for seven users.

The executive routine consists of two sections as shown in Figure 4. The common portion remains perpetually in core memory and contains the buffers for characters that are being transmitted to each console or punch, and characters typed in en route to a users' program. The common portion contains the routines that service input/output devices as required, and the program sequencing algorithm. The private block of the executive routine contains routines that provide the input/output facility status information and features specific to a particular user. In general, the private block is different for each user in correspondence with differing needs for services and input/output devices.

The private block is always transferred to and from the

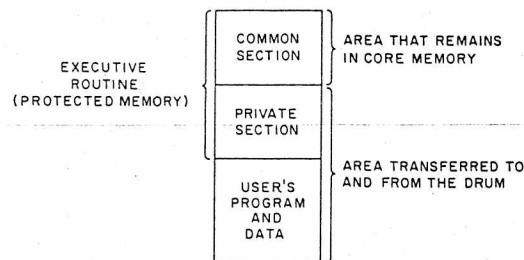


FIG. 4. Organization of executive routine in memory

drum with the user's program when a program swap occurs.

The executive routine operates with the machine in an executive mode which is entered through a trap or an executive interrupt.

A trap arises from any of a number of specific instructions encountered during operation of a user's program, which include the following.

1. Illegal instructions—the halt instruction, instructions having addresses above the memory bound, instructions referring to in/out devices not assigned to this user, and special instructions reserved for the executive routine.
2. Input/output instructions interpreted by the executive routine—instructions operating the console typewriters, tape punch and reader, and drum instructions.
3. Service request—a special instruction that requests an in/out assignment or a programming feature from the administrator.

When a trap occurs, the executive performs the function called for by the instruction causing the trap, and returns control to the user's program. In the case of illegal instructions, control is not returned to the users' program, but the administrative routine is notified so that it may take appropriate action. The organization of the executive routine is illustrated schematically in Figure 5.

Executive interrupts arise from conditions external to the user's program which require the attention of the executive routine. These are the following:

1. A console typewriter; the punch or the tape reader requires service.
2. A console switch is turned on or off, or a call button is operated.
3. The alarm clock counts out.

As indicated in Figure 5, an interrupt causes any necessary servicing of consoles to be done. Changes in console switches are noted. If the clock has rung, signaling the end

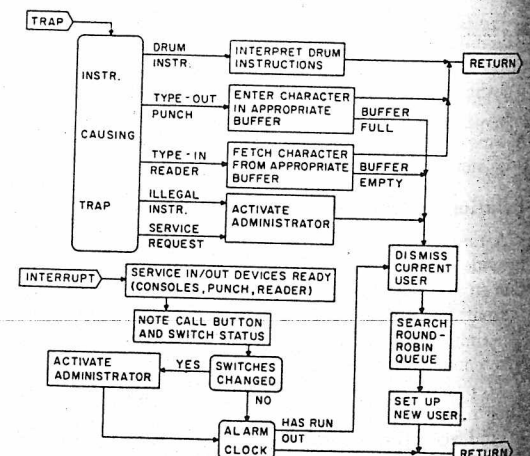


FIG. 5. Flow diagram of executive-routine operation

of a program's unit of computation, the executive routine searches for another user's program that is ready for execution.

The administrative routine normally resides on the drum and is placed in the round-robin queue when its services are required. A user may request assignment of input/output devices or services provided by the administrative routine either by typing the request at a console, or by executing a special instruction in his program.

PERFORMANCE AND SUGGESTIONS FOR IMPROVEMENT

The system described here was demonstrated in May 1963, and made available for general use during the following summer. Incorporated in the system is a version of the debugging program DDT, arranged so that it cannot be altered by any action of a user or his program. As the administrative routine is not yet complete, its functions are performed by commands to DDT.

For the activities of editing source-language programs, performing assemblies and program debugging, the time-shared facility has already proved quite valuable. In fact, most users prefer time-shared operation to informal use of the standard machine, because of features provided by the system that are not available otherwise.

One equipment-control problem has already been implemented within time-shared operation—the use of the computer to direct and track a high-gain antenna for radio astronomy experiments. This application requires the execution of a short computation once a minute throughout a run of many hours duration, and thus the advantage of time-shared operation is clear.

Although the systems programming for time-shared operation has not yet reached the form envisioned, it is still possible to foresee those aspects of system performance for which the need for improvement will be most serious.

Probably of most importance is the lack of a sufficiently large program file. Storage space on the drum is limited and is more valuable for other purposes.

It is natural at present to think of a large disk storage as a solution. We have been considering the use of the small, cheap, magnetic tape units that have recently become available as being more in keeping with the informal organization of the facility. With the magnetic tape arrangement a user could carry his personal program file to the computer in the form of a four-inch reel of tape and carry the updated file away in his pocket when he leaves.

Communication with the user through the display oscilloscope is far from ideal in the present arrangement, as the full capability of the computer is generally required to maintain an image on the display tube as a sequence of individually plotted points. Thus only one user of the system may view a quality display at a time, and the system can do little in addition except handle simple typewriter communication tasks for other users. Two techniques seem especially appropriate for improving the

display performance. One is the reduction of the data rate to the display unit and speed-up of image generation through a pattern generator that can produce characters and straight-line segments from encoded data. The other technique is letting the display unit obtain its data directly from the central memory of the computer through a data channel connection so that the attention of the processor is not required. These methods still allow the use of the light pen to select an object in a display image and interrupt the program so that the identity of the displayed object is immediately known.

With regard to communicating with laboratory equipment, the present system does not permit operation of programs that depend on execution in real time for success. Such programs cannot be run correctly without making the processing unit of the system unavailable to all other users of the system, including the executive routine, for the duration of the real-time run. A resolution of this problem is to build multiaccess computer systems with effectively more than one processing unit so that several program sequences may be executed in parallel. Then one sequence could be assigned to a real-time user for as long as necessary, while the other program sequences serve other users and executive functions.

Acknowledgments. Many persons have contributed to the success of the multiuser computer facility described here. Discussions of the M.I.T. Long Range Computation Study Committee had a great influence on this project, as did work on a similar system developed at Bolt, Beranek and Newman by McCarthy, Fredkin and Boilen [1]. We are indebted to many students for ideas and criticism throughout the course of the work. In particular, Robert Wagner suggested the scheme finally adopted for executive operation of the console typewriters, and an analysis by Thomas Hastings was valuable in estimating its performance. John Yates detailed the special logic for the system [3] and Natalio Kerlnevech assisted in its implementation and checkout. The systems programming for time-shared operation was largely done by Michael Wolfberg and Thomas Eggers. John McKenzie, Ralph Butler and John Connolly are thanked for their technical assistance in assembling and checking the hardware.

RECEIVED NOVEMBER, 1963

REFERENCES

1. BOILEN, S., FREDKIN, E., LICKLIDER, J. C. R., and MCCARTHY, J. A time-sharing debugging system for a small computer. *Proc. 1963 Spring Joint Comput. Conf.*, Vol. 23, Spartan Books, Baltimore, 1963, 51-57.
2. Report of the Long Range Computation Study Group. Massachusetts Institute of Technology, Cambridge, Mass., April 1961.
3. YATES, J. E. A time sharing system for the PDP-1 computer. Rep. AFCL-62-519, Electronic Systems Lab. M.I.T., Cambridge, Mass., June 1962.
4. M.I.T. Computation Center, *The Compatible Time-Sharing System—A Programmer's Guide*. The M.I.T. Press, Cambridge, Mass., 1963.

AFIPS

~~1962~~ 1962
Spring Joint

AN EXPERIMENTAL TIME-SHARING SYSTEM

Fernando J. Corbató, Marjorie Merwin-Daggett, Robert C. Daley

Computer Center, Massachusetts Institute of Technology

Cambridge, Massachusetts

335

Summary

It is the purpose of this paper to discuss briefly the need for time-sharing, some of the implementation problems, an experimental time-sharing system which has been developed for the contemporary IBM 7090, and finally a scheduling algorithm of one of us (FJC) that illustrates some of the techniques which may be employed to enhance and be analyzed for the performance limits of such a time-sharing system.

Introduction

The last dozen years of computer usage have seen great strides. In the early 1950's, the problems solved were largely in the construction and maintenance of hardware; in the mid-1950's, the usage languages were greatly improved with the advent of compilers; now in the early 1960's, we are in the midst of a third major modification to computer usage: the improvement of man-machine interaction by a process called time-sharing.

Much of the time-sharing philosophy, expressed in this paper, has been developed in conjunction with the work of an MIT preliminary study committee, chaired by H. Teager, which examined the long range computational needs of the Institute, and a subsequent MIT computer working committee, chaired by J. McCarthy. However, the views and conclusions expressed in this paper should be taken as solely those of the present authors.

Before proceeding further, it is best to give a more precise interpretation to time-sharing. One can mean using different parts of the hardware at the same time for different tasks, or one can mean several persons making use of the computer at the same time. The first meaning, often called multiprogramming, is oriented towards hardware efficiency in the sense of attempting to attain complete utilization of all components^{5,6,7,8}. The second meaning of time-sharing, which is meant here, is primarily concerned with the efficiency of persons trying to use a computer^{1,2,3,4}. Computer efficiency should still be considered but only in the perspective of the total system utility.

The motivation for time-shared computer usage arises out of the slow man-computer interaction rate presently possible with the bigger, more advanced computers. This rate has changed little (and has become worse in some cases) in the last decade of widespread computer use.¹⁰

In part, this effect has been due to the fact that as elementary problems become mastered on the computer, more complex problems immediately become of interest. As a result, larger and more complicated programs are written to take advantage of larger and faster computers. This process inevitably leads to more programming errors and a longer period of time required for debugging. Using current batch monitor techniques, as is done on most large computers, each program bug usually requires several hours to eliminate, if not a complete day. The only alternative presently available is for the programmer to attempt to debug directly at the computer, a process which is grossly wasteful of computer time and hampered seriously by the poor console communication usually available. Even if a typewriter is the console, there are usually lacking the sophisticated query and response programs which are vitally necessary to allow effective interaction. Thus, what is desired is to drastically increase the rate of interaction between the programmer and the computer without large economic loss and also to make each interaction more meaningful by extensive and complex system programming to assist in the man-computer communication.

To solve these interaction problems we would like to have a computer made simultaneously available to many users in a manner somewhat like a telephone exchange. Each user would be able to use a console at his own pace and without concern for the activity of others using the system. This console could as a minimum be merely a typewriter but more ideally would contain an incrementally modifiable self-sustaining display. In any case, data transmission requirements should be such that it would be no major obstacle to have remote installation from the computer proper.

The basic technique for a time-sharing system is to have many persons simultaneously using the computer through typewriter consoles with a time-sharing supervisor program sequentially running each user program in a short burst or quantum of computation. This sequence, which in the most straightforward case is a simple round-robin, should occur often enough so that each user program which is kept in the high-speed memory is run for a quantum at least once during each approximate human reaction time ($\sim .2$ seconds). In this way, each user sees a computer fully responsive to even single key strokes each of which may require only trivial computation; in the non-trivial cases, the user sees a gradual reduction of the response time which is proportional to the complexity of the response calculation, the slowness of the computer, and the total number of active users. It should be clear, however, that if there are n users actively requesting service at one time, each user will only see on the average $1/n$ of the effective computer speed. During the period of high interaction rates while debugging programs, this should not be a hindrance since ordinarily the required amount of computation needed for each debugging computer response is small compared to the ultimate production need.

Not only would such a time-sharing system improve the ability to program in the conventional manner by one or two orders of magnitude, but there would be opened up several new forms of computer usage. There would be a gradual reformulation of many scientific and engineering applications so that programs containing decision trees which currently must be specified in advance would be eliminated and instead the particular decision branches would be specified only as needed. Another important area is that of teaching machines which, although frequently trivial computationally, could naturally exploit the consoles of a time-sharing system with the additional bonus that more elaborate and adaptive teaching programs could be used. Finally, as attested by the many small business computers, there are numerous applications in business and in industry where it would be advantageous to have powerful computing facilities available at isolated locations with only the incremental capital investment of each console. But it is important to realize that even without the above and other new applications, the major advance in programming intimacy available from time-sharing would be of immediate value to computer installations in universities, research laboratories, and engineering firms where program debugging is a major problem.

Implementation Problems

As indicated, a straightforward plan for time-sharing is to execute user programs for small quanta of computation without priority in a simple round-robin; the strategy of time-sharing can be more complex as will be shown later, but the above simple scheme is an adequate solution. There are still many problems, however, some best solved by hardware, others affecting the programming conventions and practices. A few of the more obvious problems are summarized:

Hardware Problems:

1. Different user programs if simultaneously in core memory may interfere with each other or the supervisor program so some form of memory protection mode should be available when operating user programs.

2. The time-sharing supervisor may need at different times to run a particular program from several locations. (Loading relocation bits are no help since the supervisor does not know how to relocate the accumulator, etc.) Dynamic relocation of all memory accesses that pick up instructions or data words is one effective solution.

3. Input-output equipment may be initiated by a user and read words in on another user program. A way to avoid this is to trap all input-output instructions issued by a user's program when operated in the memory protection mode.

4. A large random-access back-up storage is desirable for general program storage files for all users. Present large capacity disc units appear to be adequate.

5. The time-sharing supervisor must be able to interrupt a user's program after a quantum of computation. A program-initiated one-shot multivibrator which generates an interrupt a fixed time later is adequate.

6. Large core memories (e.g. a million words) would ease the system programming complications immensely since the different active user programs as well as the frequently used system programs such as compilers, query programs, etc. could remain in core memory at all times.

Programming Problems:

1. The supervisor program must do automatic user usage charge accounting. In general,

the user should be charged on the basis of a system usage formula or algorithm which should include such factors as computation time, amount of high-speed memory required, rent of secondary memory storage, etc.

2. The supervisor program should coordinate all user input-output since it is not desirable to require a user program to remain constantly in memory during input-output limited operations. In addition, the supervisor must coordinate all usage of the central, shared high-speed input-output units serving all users as well as the clocks, disc units, etc.

3. The system programs available must be potent enough so that the user can think about his problem and not be hampered by coding details or typographical mistakes. Thus, compilers, query programs, post-mortem programs, loaders, and good editing programs are essential.

4. As much as possible, the users should be allowed the maximum programming flexibility both in choices of language and in the absence of restrictions.

Usage Problems

1. Too large a computation or excessive typewriter output may be inadvertently requested so that a special termination signal should be available to the user.

2. Since real-time is not computer usage-time, the supervisor must keep each user informed so that he can use his judgment regarding loops, etc.

3. Computer processor, memory and tape malfunctions must be expected. Basic operational questions such as "Which program is running?" must be answerable and recovery procedures fully anticipated.

An Experimental Time-Sharing System for the IBM 7090

Having briefly stated a desirable time-sharing performance, it is pertinent to ask what level of performance can be achieved with existent equipment. To begin to answer this question and to explore all the programming and operational aspects, an experimental time-sharing system has been developed. This system was originally written for the IBM 709 but has since been converted for use with the 7090 computer.

The 7090 of the MIT Computation Center has, in addition to three channels with 19 tape units, a fourth channel with the standard Direct Data Connection. Attached to the Direct Data Connection is a real-time equipment buffer and control rack designed and built under the direction of H. Teager and his group. This rack has a variety of devices attached but the only ones required by the present systems are three flexowriter typewriters. Also installed on the 7090 are two special modifications (i.e. RPO's): a standard 60 cycle accounting and interrupt clock, and a special mode which allows memory protection, dynamic relocation and trapping of all user attempts to initiate input-output instructions.

In the present system the time-sharing occurs between four users, three of whom are on-line each at a typewriter in a foreground system, and a fourth passive user of the background Fap-Mad-Madtran-BSS Monitor System similar to the Fortran-Fap-BSS Monitor System (FMS) used by most of the Center programmers and by many other 7090 installations.

Significant design features of the foreground system are:

1. It allows the user to develop programs in languages compatible with the background system,

2. Develop a private file of programs,

3. Start debugging sessions at the state of the previous session, and

4. Set his own pace with little waste of computer time.

Core storage is allocated such that all users operate in the upper 27,000 words with the time-sharing supervisor (TSS) permanently in the lower 5,000 words. To avoid memory allocation clashes, protect users from one another, and simplify the initial 709 system organization, only one user was kept in core memory at a time. However, with the special memory protection and relocation feature of the 7090, more sophisticated storage allocation procedures are being implemented. In any case, user swaps are minimized by using 2-channel overlapped magnetic tape reading and writing of the pertinent locations in the two user programs.

The foreground system is organized around commands that each user can give on his typewriter and the user's private program files which presently (for want of a disc unit) are kept on a separate magnetic tape for each user.

* This group is presently using another approach⁹ in developing a time-sharing system for the MIT 7090.

For convenience the format of the private tape files is such that they are card images, have title cards with name and class designators and can be written or punched using the off-line equipment. (The latter feature also offers a crude form of large-scale input-output.) The magnetic tape requirements of the system are the seven tapes required for the normal functions of the background system, a system tape for the time-sharing supervisor that contains most of the command programs, and a private file tape and dump tape for each of the three foreground users.

The commands are typed by the user to the time-sharing supervisor (not to his own program) and thus can be initiated at any time regardless of the particular user program in memory. For similar coordination reasons, the supervisor handles all input-output of the foreground system typewriters. Commands are composed of segments separated by vertical strokes; the first segment is the command name and the remaining segments are parameters pertinent to the command. Each segment consists of the last 6 characters typed (starting with an implicit 6 blanks) so that spacing is an easy way to correct a typing mistake. A carriage return is the signal which initiates action on the command. Whenever a command is received by the supervisor, "WAIT", is typed back followed by "READY." when the command is completed. (The computer responses are always in the opposite color from the user's typing.) While typing, an incomplete command line may be ignored by the "quit" sequence of a code delete signal followed by a carriage return. Similarly after a command is initiated, it may be abandoned if a "quit" sequence is given. In addition, during unwanted command typeouts, the command and output may be terminated by pushing a special "stop output" button.

The use of the foreground system is initiated whenever a typewriter user completes a command line and is placed in a waiting command queue. Upon completion of each quantum, the time-sharing supervisor gives top priority to initiating any waiting commands. The system programs corresponding to most of the commands are kept on the special supervisor command system tape so that to avoid waste of computer time, the supervisor continues to operate the last user program until the desired command program on tape is positioned for reading. At this point, the last user is read out on his dump tape, the command program read in, placed in a working status and initiated as a new user program. However, before starting the new user for a quantum of computation, the supervisor again checks for any waiting command of another user and if necessary begins the look-ahead positioning of the command system tape while operating the new user.

Whenever the waiting command queue is empty, the supervisor proceeds to execute a simple round-robin of those foreground user programs in the working status queue. Finally, if both these queues are empty, the background user program is brought in and run a quantum at a time until further foreground system actively develops.

Foreground user programs leave the working status queue by two means. If the program proceeds to completion, it can reenter the supervisor in a way which eliminates itself and places the user in dead status; alternatively, by a different entry the program can be placed in a dormant status (or be manually placed by the user executing a quit sequence). The dormant status differs from the dead status in that the user may still restart or examine his program.

User input-output is through each type-writer, and even though the supervisor has a few lines of buffer space available, it is possible to become input-output limited. Consequently, there is an additional input-output wait status, similar to the dormant, which the user is automatically placed in by the supervisor program whenever input-output delays develop. When buffers become near empty on output or near full on input, the user program is automatically returned to the working status; thus waste of computer time is avoided.

Commands

To clarify the scope of the foreground system and to indicate the basic tools available to the user, a list of the important commands follows along with brief summaries of their operations:

1. | α

α = arbitrary text treated as a comment.

2. login | α | β

α = user problem number
 β = user programmer number

Should be given at beginning of each user's session. Rewinds user's private file tape; clears time accounting records.

3. logout

Should be given at end of each user's session. Rewinds user's private file tape; punches on-line time accounting cards.

4. input

Sets user in input mode and initiates automatic generation of line numbers. The user

types a card image per line according to a format appropriate for the programming language. (The supervisor collects these card images at the end of the user's private file tape.) When in the automatic input mode, the manual mode may be entered by giving an initial carriage return and typing the appropriate line number followed by | and line for as many lines as desired. To reenter the automatic mode, an initial carriage return is given.

The manual mode allows the user to overwrite previous lines and to insert lines. (cf. File Command.)

5. edit | α | β

α = title of file
 β = class of file

The user is set in the automatic input mode with the designated file treated as initial input lines. The same conventions apply as to the input command.

6. file | α | β

α = title to be given to file
 β = class of language used during input

The created file will consist of the numbered input lines (i.e. those at the end of the user's private file tape) in sequence; in the case of duplicate line numbers, the last version will be used. The line numbers will be written as sequence numbers in the corresponding card images of the file.

For convenience the following editing conventions apply to input lines:

- an underline signifies the deletion of the previous characters of the line.
- a backspace signifies the deletion of the previous character in the field.

The following formats apply:

- FAP: symbol, tab, operation, tab, variable field and comment.
- MAD, MADTRAN, FORTRAN: statement label, tab, statement. To place a character in the continuation column: statement label, tab, backspace, character, statement.
- DATA: cols. 1-72.

7. fap | α

Causes the file designated as α , fap to be translated by the FAP translator (assembler). Files α , symtb and α ,bss are added to the user's private file tape giving the symbol table and the relocatable binary BSS form of the file.

8. mad | α

Causes file α ,mad to be translated by the MADtranslator (compiler). File α ,bss is created.

9. madtrn | α

Causes file α ,madtrn (i.e. a pseudo-Fortran language file) to be edited into an equivalent file α ,mad (added to the user's file) and translation occurs as if the command mad| α had been given.

10. load | α_1 | α_2 | ... | α_n

Causes the consecutive loading of files α_i ,bss (i=1,2,...,n). An exception occurs if α_1 = (libe), in which case file α_{i+1} ,bss is searched as a library file for all subprograms still missing. (There can be further library files.)

11. use | α_1 | α_2 | ... | α_n

This command is used whenever a load or previous use command notifies the user of an incomplete set of subprograms. Same α_i conventions as for load.

12. start | α | β

Starts the program setup by the load and use commands (or a dormant program) after first positioning the user private file tape in front of the title card for file α , β . (If β is not given, a class of data is assumed; if both α and β are not given, no tape movement occurs and the program is started.)

13. pm | α

α = "lights", "stomap", or the usual format of the standard Center post-mortem (F2PM) request: subprogram name | loc₁ | loc₂ | mode | direction where mode and direction are optional.

Produces post-mortem of user's dormant program according to request specified by α . (E.g. matrix | 5 | 209 | flo | rev will cause to be printed on the user's typewriter the contents of subprogram "matrix" from relative locations 5 to 209 in floating point form and in reverse sequence.)

14. skippm

Used if a pm command is "quit" during output and the previous program interruption is to be restarted.

15. listf

Types out list of all file titles on user's private file tape.

16. printf | α | β | γ

Types out file α, β starting at line number γ . If γ is omitted, the initial line is assumed. Whenever the user's output buffer fills, the command program goes into an I/O wait status allowing other users to time-share until the buffer needs refilling.

17. xdump | α | β

Creates file α, β (if β omitted, xdump assumed) on user's private file tape consisting of the complete state of the user's last dormant program.

18. xdump | α | β

Inverse of xdump command in that it resets file α, β as the user's program, starting it where it last left off.

Although experience with the system to date is quite limited, first indications are that programmers would readily use such a system if it were generally available. It is useful to ask, now that there is some operating experience with the 7090 system, what observations can be made. An immediate comment is that once a user gets accustomed to computer response, delays of even a fraction of a minute are exasperatingly long, an effect analogous to conversing with a slow-speaking person. Similarly, the requirement that a complete typewritten line rather than each character be the minimum unit of man-computer communication is an inhibiting factor in the sense that a press-to-talk radio-telephone conversation is more stilted than that of an ordinary telephone. Since maintaining a rapid computer response on a character by character basis requires at least a vestigial response program in core memory at all times, the straightforward solution within the present system is to have more core memory available. At the very least, an extra bank of memory for the time-sharing supervisor would ease compatibility problems with programs already written for 32,000 word 7090's.

For reasons of expediency, the weakest portions of the present system are the conventions for input, editing of user files, and the degree of rapid interaction and intimacy possible while debugging. Since to a large extent these areas involve the taste, habits, and psychology of the users, it is felt that proper solutions will require considerable experimentation and pragmatic evaluation; it is also clear that these areas cannot be treated in the abstract for the programming languages used will influence greatly the appropriate techniques. A greater use of symbolic referencing for locations, program names and variables is certainly desired; symbolic post-mortem programs, trace programs, and before-and-after differential dump programs should play useful roles in the debugging procedures.

In the design of the present system, great care went into making each user independent of the other users. However, it would be a useful extension of the system if this were not always the case. In particular, when several consoles are used in a computer controlled group such as in management or war games, in group behavior studies, or possibly in teaching machines, it would be desirable to have all the consoles communicating with a single program.

Another area for further improvement within the present system is that of file maintenance, since the presently used tape units are a hindrance to the easy deletion of user program files. Disc units will be of help in this area as well as with the problem of consolidating and scheduling large-scale central input-output generated by the many console users.

Finally, it is felt that it would be desirable to have the distinction between the foreground and background systems eliminated. The present-day computer operator would assume the role of a stand-in for the background users, using an operator console much like the other user consoles in the system, mounting and demounting magnetic tapes as requested by the supervisor, receiving instructions to read card decks into the central disc unit, etc. Similarly the foreground user, when satisfied with his program, would by means of his console and the supervisor program enter his program into the queue of production background work to be performed. With these procedures implemented the distinction of whether one is time-sharing or not would vanish and the computer user would be free to choose in an interchangeable way that mode of operation which he found more suitable at a particular time.

A Multi-Level Scheduling Algorithm

Regardless of whether one has a million word core memory or a 32,000 word memory as currently exists on the 7090, one is inevitably faced with the problem of system saturation where the total size of active user programs exceeds that of the high-speed memory or there are too many active user programs to maintain an adequate response at each user console. These conditions can easily arise with even a few users if some of the user programs are excessive in size or in time requirements. The predicament can be alleviated if it is assumed that a good design for the system is to have a saturation procedure which gives graceful degradation of the response time and effective real-time computation speed of the large and long-running users.

To show the general problem, Figure 1 qualitatively gives the user service as a function of n , the number of active users. This service parameter might be either of the two key factors: computer response time or n times the real-time computation speed. In either case there is some critical number of active users, N , representing the effective user capacity, which causes saturation. If the strategy near saturation is to execute the simple round-robin of all users, then there is an abrupt collapse of service due to the sudden onset of the large amount of time required to swap programs in-and-out of the secondary memory such as a disc or drum unit. Of course, Figure 1 is quite qualitative since it depends critically on the spectrum of user program sizes as well as the spectrum of user operating times.

To illustrate the strategy that can be employed to improve the saturation performance of a time-sharing system, a multi-level scheduling algorithm is presented. This algorithm also can be analyzed to give broad bounds on the system performance.

The basis of the multi-level scheduling algorithm is to assign each user program as it enters the system to be run (or completes a response to a user) to an l th level priority queue. Programs are initially entered into a level l_0 , corresponding to their size such that

$$l_0 = \left\lceil \log_2 \left(\left\lceil \frac{w}{w_q} \right\rceil + 1 \right) \right\rceil \quad (1)$$

where w is the number of words in the program, w_q is the number of words which can be transmitted in and out of the high-speed memory from the secondary memory in the time of one quantum, q , and the bracket indicates "the integral part of". Ordinarily the time of a quantum, being the basic time unit, should be as small as possible without excessive overhead losses when the supervisor switches from one program in high-speed memory to another. The process starts with the time-sharing supervisor operating the program at the head of the lowest level occupied queue, l , for up to 2^l quanta of time and then if the program is not completed (i.e. has not made a response to the user) placing it at the end of the $l+1$ level queue. If there are no programs entering the system at levels lower than l , this process proceeds until the queue at level l is exhausted; the process is then iteratively begun again at level $l+1$, where now each program is run for 2^{l+1} quanta of time. If during the execution of the 2^l quanta of a program at level l , a lower level, l' , becomes occupied, the current user is replaced at the head of the l' th queue and the process is reinitiated at level l' .

Similarly, if a program of size w at level l , during operation requests a change in memory size from the time-sharing supervisor, then the enlarged (or reduced) version of the program should be placed at the end of the l' queue where

$$l' = l + \left\lceil \log_2 \left(\left\lceil \frac{w_p}{w_q} \right\rceil + 1 \right) \right\rceil \quad (2)$$

Again the process is re-initiated with the head-of-the-queue user at the lowest occupied level of l' .

Several important conclusions can be drawn from the above algorithm which allow the performance of the system to be bounded.

Computational Efficiency

1. Because a program is always operated for a time greater than or equal to the swap time (i.e. the time required to move the program in and out of secondary memory), it follows that the computational efficiency never falls below one-half. (Clearly, this fraction is adjustable in the formula for the initial level, l_0 .) An alternative way of viewing this bound is to say that the real-time computing speed available to one out of n active users is no worse than if there were $2n$ active users all of whose programs were in the high-speed memory.

Response Time

2. If the maximum number of active users is N , then an individual user of a given program size can be guaranteed a response time,

$$t_r \leq 2Nq \left(\left\lceil \frac{w_p}{w_q} \right\rceil + 1 \right) \quad (3)$$

since the worst case occurs when all competing user programs are at the same level. Conversely, if t_r is a guaranteed response of arbitrary value and the largest size of program is assumed, then the maximum permissible number of active users is bounded.

Long Runs

3. The relative swap time on long runs can be made vanishingly small. This conclusion follows since the longer a program is run, the higher the level number it cascades to with a correspondingly smaller relative swap time. It is an important feature of the algorithm that long runs must in effect prove they are long so that programs which have an unexpected demise are detected quickly. In order that there be a finite number of levels, a maximum level number, L , can be established such that the asymptotic swap overhead is some arbitrarily small percentage, p :

$$L = \left\lceil \log_2 \left(\left\lceil \frac{w_{pmax}}{p_{wq}} \right\rceil + 1 \right) \right\rceil \quad (4)$$

where w_{pmax} is the size of the largest possible program.

Multi-level vs. Single-level Response Times

4. The response time for programs of equal size, entering the system at the same time, and being run for multiple quanta, is no worse than approximately twice the response-time occurring in a single quanta round-robin procedure. If there are n equal sized programs started in a queue at level l , then the worst case is that of the end-of-the-queue program which is ready to respond at the very first quantum run at the $(l+j)$ level. Using the multi-level algorithm, the total delay for the end-of-the-queue program is by virtue of the geometric series of quanta:

$$T_m \sim q 2^j \{n(2^j-1) + (n-1) 2^j\} \quad (5)$$

Since the end-of-the-queue user has computed for a time of $2^j(2^j-1)$ quanta, the equivalent single-level round-robin delay before a response is:

$$T_s \sim q 2^j \{n(2^j-1)\}. \quad (6)$$

Hence

$$\frac{T_m}{T_s} \sim 1 + \left(\frac{n-1}{n} \right) \left(\frac{2^j}{2^j-1} \right) \sim 2 \quad (7)$$

and the assertion is shown. It should be noted that the above conditions, where program swap times are omitted, which are pertinent when all programs remain in high-speed memory, are the least favorable for the multi-level algorithm; if swap times are included in the above analysis, the ratio of T_m/T_s can only become smaller and may become much less than unity. By a similar analysis it is easy to show that even in the unfavorable case where there are no program swaps, head-of-the-queue programs that terminate just as the 2^{j+1} quanta are completed receive under the multi-level algorithm a response which is twice as fast as that under the single-level round-robin (i.e. $T_m/T_s = 1/2$).

Highest Served Level

5. In the multi-level algorithm the level classification procedure for programs is entirely automatic, depending on performance and program size rather than on the declarations (or hopes) of each user. As a user taxes the system, the degradation of service occurs progressively starting with the higher level users of either large or long-running programs; however, at some level no user programs may be run because of too many active users at lower levels. To determine

a bound on this cut-off point we consider N active users at level l each running 2^j quanta, terminating, and reentering the system again at level l at a user response time, t_u , later. If there is to be no service at level $l+1$, then the computing time, $Nq2^j$, must be greater than or equal to t_u . Thus the guaranteed active levels, l_a , are given by the relation:

$$l_a \leq \left\lceil \log_2 \left(\frac{t_u}{Nq} \right) \right\rceil \quad (8)$$

In the limit, t_u could be as small as a minimum user reaction time (~ 2 sec.), but the expected value would be several orders of magnitude greater as a result of the statistics of a large number of users.

The multi-level algorithm as formulated above makes no explicit consideration of the seek or latency time required before transmission of programs to and from disc or drum units when they are used as the secondary memory, (although formally the factor w_q could contain an average figure for these times). One simple modification to the algorithm which usually avoids wasting the seek or latency time is to continue to operate the last user program for as many quanta as are required to ready the swap of the new user with the least priority user; since ordinarily only the higher level number programs would be forced out into the secondary memory, the extended quanta of operation of the old user while seeking the new user should be but a minor distortion of the basic algorithm.

Further complexities are possible when the hardware is appropriate. In computers with input-output channels and low transmission rates to and from secondary memory, it is possible to overlap the reading and writing of the new and old users in and out of high-speed memory while operating the current user. The effect is equivalent to using a drum giving 100% multiplexor usage but there are two liabilities, namely, no individual user can utilize all the available user memory space and the look-ahead procedure breaks down whenever an unanticipated scheduling change occurs (e.g. a program terminates or a higher-priority user program is initiated).

Complexity is also possible in storage allocation but certainly an elementary procedure and a desirable one with a low-transmission rate secondary memory is to consolidate in a single block all high-priority user programs whenever sufficient fragmentary unused memory space is available to read in a new user program. Such a procedure is indicated in the flow diagram of the multi-level scheduling algorithm which is given as Figure 2.

It should also be noted that Figure 2 only accounts for the scheduling of programs in a working status and still does not take into account the storage allocation of programs which are in a dormant (or input-output wait status). One systematic method of handling this case is to modify the scheduling algorithm so that programs which become dormant at level l are entered into the queue at level $l+1$. The scheduling algorithm proceeds as before with the dormant programs continuing to cascade but not operating when they reached the head of a queue. Whenever a program must be removed from high-speed memory, a program is selected from the end-of-the-queue of the highest occupied level number.

Finally, it is illuminating to apply the multi-level scheduling algorithm bounds to the contemporary IBM 7090. The following approximate values are obtained:

$q = 16$ m.s. (based on 1% switching overhead)

$w_q = 120$ words (based on one IBM 1301 model 2 disc unit without seek or latency times included)

$t_r \leq 8Nf_{sec.}$ (based on programs of (32k)f words)

$l_a \leq \log_2 (1000/N)$ (based on $t_u = 16$ sec.)

$l_0 \leq 8$ (based on a maximum program size of 32K words)

Using the arbitrary criteria that programs up to the maximum size of 32,000 words should always get some service, which is to say that $\max l_a = \max l_0$, we deduce as a conservative estimate that N can be 4 and that at worst the response time for a trivial reply will be 32 seconds.

The small value of N arrived at is a direct consequence of the small value of w_q that results from the slow disc word transmission rate. This rate is only 3.3% of the maximum core memory multiplexor rate. It is of interest that using high-capacity high-speed drums of current design such as in the Sage System of in the IBM Sabre System it would be possible to attain nearly 100% multiplexor utilization and thus multiply w_q by a factor of 30. It immediately follows that user response times equivalent to those given above with the disc unit would be given to 30 times as many persons or to 120 users; the total computational capacity, however, would not change.

In any case, considerable caution should be used with capacity and computer response time estimates since they are critically dependent upon the distribution functions for the user response time, t_u , and the user program size, w_p , and the computational capacity requested by each user. Past experience using conventional programming systems is of little assistance because these distribution functions will depend very strongly upon the programming systems made available to the time-sharing users as well as upon the user habit patterns which will gradually evolve.

Conclusions

In conclusion, it is clear that contemporary computers and hardware are sufficient to allow moderate performance time-sharing for a limited number of users. There are several problems which can be solved by careful hardware design, but there are also a large number of intricate system programs which must be written before one has an adequate time-sharing system. An important aspect of any future time-shared computer is that until the system programming is completed, especially the critical time-sharing supervisor, the computer is completely worthless. Thus, it is essential for future system design and implementation that all aspects of time-sharing system problems be explored and understood in prototype form on present computers so that major advances in computer organization and usage can be made.

Acknowledgements

The authors wish to thank Bernard Galler, Robert Graham and Bruce Arden, of the University of Michigan, for making the MAD compiler available and for their advice with regard to its adaption into the present time-sharing system. The version of the Madtran Fortran-to-Mad editor program was generously supplied by Robert Rosin of the University of Michigan. Of the MIT Computation Center staff, Robert Creasy was of assistance in the evaluation of time-sharing performance, Lynda Korn is to be credited for her contributions to the pm and madtran commands, and Evelyn Dow for her work on the fap command.

References

1. Strachey, C., "Time Sharing in Large Fast Computers," Proceedings of the International Conference on Information Processing, UNESCO (June, 1959), Paper B.2.19.

2. Licklider, J. C. R., "Man-Computer Symbiosis," *IRE Transactions on Human Factors in Electronics*, HFE-1, No. 1 (March, 1960), 4-11.

3. Brown, G., Licklider, J. C. R., McCarthy, J., and Perlis, A., lectures given spring, 1961, Management and the Computer of the Future, (to be published by the M.I.T. Press, March, 1962).

4. Corbato, F. J., "An Experimental Time-Sharing System," *Proceedings of the IBM University Director's Conference*, July, 1961 (to be published).

5. Schmitt, W. F., Tonik, A. B., "Symptomatically Programmed Computers," *Proceedings of the International Conference on Information Processing*, UNESCO, (June, 1959) Paper B.2.18.

6. Codd, E. F., "Multiprogram Scheduling," *Communications of the ACM*, 3, 6 (June, 1960), 347-350.

7. Heller, J., "Sequencing Aspects of Multiprogramming," *Journal of the ACM*, 8, 3 (July, 1961), 426-439.

8. Leeds, H. D., Weinberg, G. M., "Multiprogramming," *Computer Programming Fundamentals*, 356-359, McGraw-Hill (1961).

9. Teager, H. M., "Real-Time Time-Shared Computer Project," *Communications of the ACM*, 5, 1 (January, 1962) Research Summaries, 62.

10. Teager, H. M., McCarthy, J., "Time-Shared Program Testing," paper delivered at the 14th National Meeting of the ACM (not published).

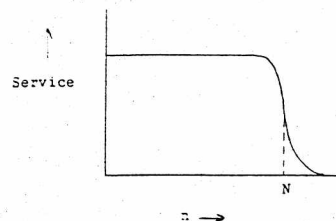


Figure 1. Service vs. Number of Active Users

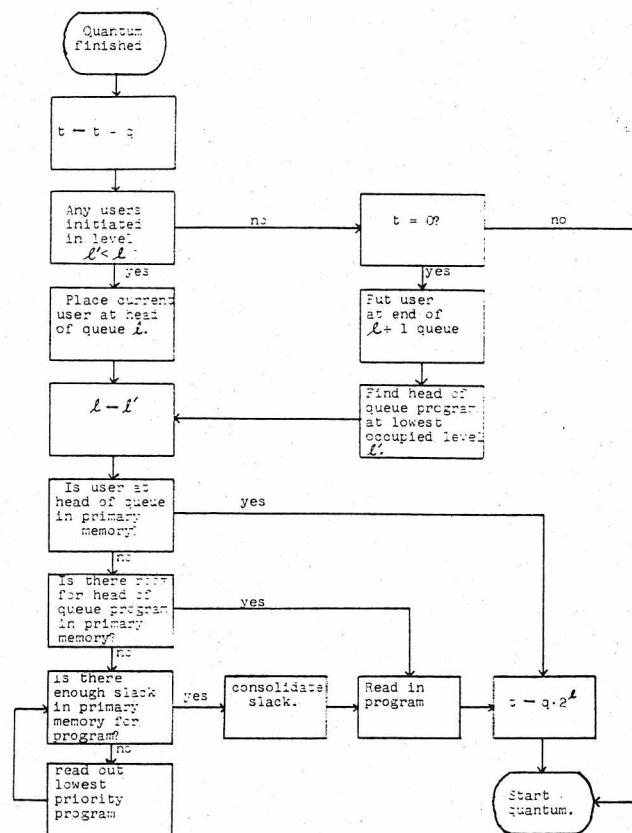


Figure 2. Flow Chart of Multi-Level Scheduling Algorithm

A PROGRAMMING LANGUAGE

Kenneth E. Iverson

Research Division, IBM Corporation
Yorktown Heights, New York

The paper describes a succinct problem-oriented programming language. The language is broad in scope, having been developed for, and applied effectively in, such diverse areas as microprogramming, switching theory, operations research, information retrieval, sorting theory, structure of compilers, search procedures, and language translation. The language permits a high degree of useful formalism. It relies heavily on a systematic extension of a small set of basic operations to vectors, matrices, and trees, and on a family of flexible selection operations controlled by logical vectors. Illustrations are drawn from a variety of applications.

The programming language outlined here has been designed for the succinct description of algorithms. The intended range of algorithms is broad; effective applications include microprogramming, switching theory, operations research, information retrieval, sorting theory, structure of compilers, search procedures, and language translation. The symbols used have been chosen so as to be concise and mnemonic, and their choice has not been restricted to the character sets provided in contemporary printers and computers. A high degree of formalism is provided.

Basic Operations

The language is based on a consistent unification and extension of existing mathematical notations, and upon a systematic extension of a small set of basic arithmetic and logical operations to vectors, matrices, and trees. The arithmetic operations include the four familiar arithmetic operations and the absolute value (denoted by the usual symbols) as well as the floor, ceiling, and residue functions denoted and defined as follows:

Name	Symbol	Definition
floor	$\lfloor x \rfloor$	$\lfloor x \rfloor \leq x < \lfloor x \rfloor + 1$
ceiling	$\lceil x \rceil$	$\lceil x \rceil \geq x > \lceil x \rceil - 1$
residue mod m	$m n$	$n = mq + r; 0 \leq r < m$

where $\lfloor x \rfloor$, $\lceil x \rceil$, m , n , and q are integers.

The logical operations and, or, and not are defined upon the logical variables 0 and 1 and are denoted by $\wedge$, $\vee$, and an overbar. They are augmented by the relational statement (proposition) (xRy) defined as follows. If x and y are any

entities (e.g., numerals, alphabetic literals, or logical variables) and R is any relation defined upon them, then (xRy) is equal to 1 or 0 according to whether the relation R does or does not hold between x and y . Thus $(i=j)$ is the familiar Kronecker delta δ_{ij} , $(u \neq v)$ is the exclusive or function of the logical variables u and v , and

$$\text{sgn } x = (x > 0) - (x < 0)$$

is the familiar sign function, defined as +1, 0, or -1 according as x is strictly positive, zero, or strictly negative.

Vectors and Matrices

A vector is denoted by an underlined lower-case letter (e.g., $\underline{x}$), its i -th component by $\underline{x}_i$, and its dimension by $\nu(\underline{x})$. A matrix is denoted by an underlined uppercase letter (e.g., $\underline{X}$), its i -th row by $\underline{X}^i$, its j -th column by $\underline{X}_j$, its ij -th element by $\underline{X}^i_j$, its row dimension (i.e., the common dimension of its row vectors) by $\nu(\underline{X})$, and its column dimension by $\mu(\underline{X})$.

All of the basic operations are extended component-by-component to vectors and matrices. Thus,

$$\underline{z} = \underline{x} + \underline{y} \iff \underline{z}_i = \underline{x}_i + \underline{y}_i$$

$$\underline{Z} = \underline{X} \times \underline{Y} \iff \underline{Z}^i_j = \underline{X}^i_k \times \underline{Y}^k_j$$

$$\underline{w} = \underline{u} \wedge \underline{v} \iff \underline{w}_i = \underline{u}_i \wedge \underline{v}_i$$

$$\underline{w} = (\underline{x} < \underline{y}) \iff \underline{w}_i = (\underline{x}_i < \underline{y}_i)$$

The symbol $\underline{\epsilon}(n)$ will denote a logical vector of dimension n whose components are all unity, and $\underline{0}(n)$ therefore denotes the zero vector. The